Institut für Parallele und Verteilte Systeme
Abteilung Anwendersoftware
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 2983

Einsatz unscharfer Suchstrategien für Datenbanken in betriebswirtschaftlichen Webanwendungen

Dennis Scheck

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Betreuer:	Dipl.-Inf. Alexander Moosbrugger IPVS, Universität Stuttgart
	Dr. Elke Schweizer CI/AFJ, Robert Bosch GmbH
begonnen am:	3. November 2009
beendet am:	5. Mai 2010
CR-Klassifikation:	H.3.3, H.2.8

Abstract

Zwei häufige Probleme bei datenintensiven Anwendungen werden durch die üblicherweise verwendete exakte Suche verursacht. Zum einen können unterschiedliche Schreibweisen beim Anlegen und späteren Suchen eines Datensatzes das Auffinden erschweren oder gar unmöglich machen. Zum anderen können dadurch zahlreiche Dubletten entstehen, was bei betriebswirtschaftlichen Anwendungen in großen Unternehmen viele weitere Probleme nach sich ziehen kann.

In dieser Arbeit werden verschiedene phonetische und distanzbasierte Methoden zur unscharfen Suche betrachtet und diverse Möglichkeiten des Einsatzes in Web-Applikationen evaluiert. Anhand der gewonnenen Erkenntnisse wird ein Prototyp implementiert, in dem Suche, dublettenfreie Anlage und Abgleich von Objekten in einem Stammdatensystem mit Hilfe von Hibernate Search unter Verwendung von Double Metaphone als phonetisches Verfahren und distanzbasierter Verfahren realisiert werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Beispiele

Verzeichnis der Listings

Bevor man beginnt, bedarf es der Überlegung und, sobald man überlegt hat, rechtzeitiger Ausführung.

– Sallust

1 Einleitung

Immer häufiger treten in Projekten der Abteilung CI/AFJ direkt Kundenwünsche nach einer unscharfen Suche auf oder wäre eine unscharfe Suchmethode zur Implementierung weiterer Funktionen, wie z. B. Dublettenerkennung und -vermeidung, hilfreich.

Der dringlichste Wunsch war eine unscharfe Suche über Kundendaten. "*Meyer* oder *Maier*? Oder gar *Mayer*?„ – „Nein, *Meier* ". Diese Dialoge sollten der Vergangenheit angehören.

Als Reaktion auf die immer wiederkehrenden Anfragen wurde diese Arbeit in Auftrag gegeben. Der Fokus liegt dabei allein auf der unscharfen Suche in Adressdaten, da hier andere Anforderungen gestellt und andere Algorithmen zur Verfügung stehen als z.B. für eine unscharfe Volltextsuche.

Die unscharfe Suche auf Stammdaten wird dabei in verschiedenen Anwendungsszenarien benötigt, auf die auch in dieser Arbeit eingegangen werden soll.

Zum einen sollen bei einer *Suche* eines Kunden, Lieferanten oder Mitarbeiters auch diejenigen gefunden werden, die ähnlich geschrieben werden. Damit soll einerseits vermieden werden, dass Tippfehler beim Anlegen oder beim Suchen das erneute Auffinden unmöglich machen, aber andererseits auch Daten gefunden werden, wenn man die genaue Schreibweise nicht kennt, weil man den Namen nur gehört oder in Erinnerung hat und nicht geschrieben vor sich sieht.

Ein zweiter Anwendungsfall ist die *Vermeidung von Dubletten*. Beim Anlegen von neuen Stammdaten passiert es immer wieder, dass mehrere Konten für einen Kunden angelegt werden. Auch hier stößt man wieder auf die selben Probleme wie bei der Suche. Darum soll vor der Anlage eines neuen Datensatzes unscharf überprüft werden, ob ein ähnlicher Datensatz nicht bereits schon vorhanden ist.

Der dritte zu berücksichtigende Anwendungsfall ist das *Zusammenführen von Datensätzen* aus unterschiedlichen Datenquellen. In vielen Projekten müssen Daten aus anderen Datenquellen mit Datensätzen aus der eigenen Anwendung kombiniert werden. Oft steht kein eindeutiger Schlüssel dafür zur Verfügung. Mit Hilfe der unscharfen Suche soll dennoch versucht werden, ein Mapping herzustellen.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen: Dieses Kapitel stellt die Grundlagen dieser Arbeit vor. Neben der Beschreibung und Erklärung verschiedener Algorithmen zur unscharfen Suche über Namen werden auch verwendete Technologien kurz erläutert.

Kapitel 3 – Evaluierung verschiedener Lösungsansätze: In Kapitel 3 werden verschiedene Möglichkeiten evaluiert, mit denen die unscharfe Suche für die BOSCH OpenJava Platform realisiert werden könnte, und die Auswahl der Technologie für die Implementierung getroffen.

Kapitel 4 – Implementierung: Hier wird die Implementierung der unscharfen Suche für die BOSCH OpenJava Platform und das Vorgehen bei der Erstellung des Prototyps beschrieben.

Kapitel 5 – Test und Bewertung der Implementierung: Eine Bewertung der Implementierung findet in diesem Kapitel statt.

Kapitel 6 – Zusammenfassung und Ausblick: Das letzte Kapitel gibt noch einmal einen Überblick über den Ablauf der Diplomarbeit und gibt Ideen für zukünftige Erweiterungen.

Anhang: Im Anhang finden sich Materialien im Zusammenhang mit dieser Arbeit.

Anmerkungen zu den Testdaten

Da für Testzwecke keine Daten in größerer Menge zur Verfügung standen, wurden aus einer Kunden-Datei mit personenbezogenen Daten aus einem produktiven Buchhaltungsbestand im CSV-Format die Spalten *Vorname*, *Nachname*, *Adresse* und *Ort* extrahiert, und diese mit Hilfe eines Perl-Skripts zufällig neu kombiniert.

Dadurch entstanden über 115.000 Datensätze mit fiktiven Personen, die jedoch einen realen Querschnitt über eine Adressdatenbank im europäischen Raum (mit Schwerpunkt auf deutschen Namen) bot.

Die meisten in Kapitel 2 verwendeten Beispiele waren in den Daten bereits vorhanden. Fehlende Beispiele wurden ergänzt, damit die bereits eingeführten Beispiele für Kapitel 3 zur Evaluierung von möglichen Verfahren verwendet werden können.

> Zu wissen, woran man selbst interessiert ist, ist die Voraussetzung, um andere Leute dafür zu interessieren.
>
> *– Walter Pater*

2 Grundlagen

In diesem Kapitel werden die Grundlagen zu später verwendeten oder evaluierten Technologien vorgestellt, sowie die Verfahren, die von den jeweiligen Bibliotheken verwendet werden. Dabei wird auch kurz die Geschichte und Weiterentwicklung der Algorithmen erwähnt, um beispielhaft Möglichkeiten und Ansätze für eigene Verbesserungen der Verfahren aufzuzeigen, die allerdings außerhalb des Rahmens dieser Arbeit liegen.

2.1 Phonetische Verfahren

Bei den phonetischen Verfahren werden nicht die Zeichen eines Wortes miteinander verglichen, wie z. B. bei einem String-Vergleich, sondern die Aussprache der jeweiligen Worte.

Zum Beispiel die Namen *Maier*, *Meier*, *Mayer*, *Meyer* oder gar *Major* werden allesamt als [ˈmɑɐ̯ɐ] ausgesprochen, obwohl es 5 verschiedene Zeichenketten sind. Vor allem die Namensträger einer der Varianten kennen das Problem, dass sie sehr häufig die korrekte Schreibweise ihres Namens ergänzend angeben müssen.

Bei phonetischen Verfahren, die man in die Klasse der Hashing-Algorithmen einordnen kann, wird versucht, den Klang eines Wortes (bzw. im Kontext dieser Arbeit hauptsächlich eines Namens) in einer standardisierten Form zu erfassen und zu codieren. Ergeben zwei verschiedene Namen den selben Code, kann man davon ausgehen, dass die Aussprache sehr ähnlich oder sogar identisch ist, auch wenn ein „*scharfer*“ String-Vergleich keine Übereinstimmungen bringt.

Da je nach Sprache (oder sogar Region innerhalb eines Sprachraums) Worte und Namen unterschiedlich ausgesprochen werden können, stoßen diese Verfahren an Grenzen. Es wäre denkbar, dass in einer Sprache zwei verschiedene Schreibweisen gleich ausgesprochen werden, in einer anderen Sprache aber ein großer Unterschied in der Aussprache besteht. Das jeweilige Verfahren muss also auf den Sprachraum angepasst sein.

Aber selbst dann können Probleme auftreten, da eine eindeutige Aussprache nicht immer existiert. Als Beispiel soll hier das – zugegebenermaßen etwas konstruierte – Kunstwort *ghoti* genannt werden, das gerne von Sprachkritikern angeführt wird, wenn man die Undeutigkeit einer Aussprache eines Wortes in der englischen Sprache demonstrieren will.

Nach den englischen Ausspracheregeln könnte *ghoti* zum einen naheliegend als [gəotiː], aber auch als [fɪʃ], also gleich wie das Wort *fish* ausgesprochen werden:

- „gh“ als /f/, wenn man es ausspricht wie in enough
- „o“ als /ɪ/, bei der Aussprache wie in women und
- „ti“ als /ʃ/ wie in emotion

Es könnte aber auch gar nicht, also stumm ausgesprochen werden, wenn

- „gh“ wie in night oder fight
- „o“ wie in people
- „t“ wie in ballet oder gourmet und
- „i“ wie in business

ausgesprochen werden.

Wobei das Beispiel genau genommen hinkt, da durch den Kontext, in dem die jeweiligen Buchstaben stehen, durchaus nur eine konkrete Aussprache möglich ist, und nur [gəotiː] als korrekte Aussprache in Frage kommt (vgl. [Ros00]). Dennoch eignet sich das Beispiel, um die Schwierigkeiten und Komplexität der phonetischen Verfahren aufzuzeigen.

Im Folgenden werden mehrere Verfahren vorgestellt und verglichen. Ausgehend von Soundex, dem ersten phonetischen Verfahren, werden die Verbesserungen von Mokotoff und Daitch des Verfahrens vorgestellt, sowie die Kölner Phonetik als Verfahren für den deutschsprachigen Raum. Schließlich Metaphone und die Erweiterung Double Metaphone, die deutlich komplexere Regeln haben, dafür aber sehr gute Resultate liefern.

Der Match Rating Approach, ein phonetisches Verfahren nach einem anderen Prinzip wie Soundex, wird ebenfalls kurz vorgestellt, um auch noch einen anderen Ansatz zu phonetischen Vergleichen zu demonstrieren.

2.1.1 Soundex

Soundex wurde von Robert Russel Ende des 19. Jahrhunderts als phonetisches Verfahren in den USA entwickelt, um für die Volkszählung über einen Code ähnliche

Namen zusammenfassen zu können (vgl. [Arm00, S. 14]). Soundex ist der Urvater vieler weiterer phonetischer Verfahren.

Der Algorithmus folgt dabei einfachen Regeln:

1. Der erste Buchstabe des Wortes wird direkt übernommen.
2. Danach werden bis zu 3 Ziffern angehängt, die nach folgendem Schema Buchstabe für Buchstabe ermittelt werden, beginnend mit dem zweiten Buchstaben des Ausgangswortes:
   a) Der Zahlenwert des Buchstabens nach Tabelle 2.1 wird an den bestehenden Code angehängt. Taucht der Buchstabe nicht in der Tabelle auf, wird er ignoriert.
   b) Bei doppelten Konsonanten wird nur der erste codiert, der zweite wird ignoriert.
   c) Aufeinanderfolgende Buchstaben, die den selben Code ergeben, werden ebenfalls nur einmal codiert, der zweite wird verworfen.
   d) Werden zwei Konsonanten mit dem selben Code durch einen Vokal oder Y getrennt, wird der zweite nicht verworfen.
3. Hat der Code danach weniger als 4 Stellen (1 Buchstabe und 3 Ziffern), wird rechts mit 0 aufgefüllt, bis insgesamt 4 Stellen vorhanden sind.

Buchstabe	**Code**
B P F V	1
C S K G J Q X Z (ß)	2
D T	3
L	4
M N	5
R	6

Tabelle 2.1: Soundex Codierschema

Beispiele für Soundex-Codierungen

Wie das folgende Beispiel 2.1 zeigt, werden 4 der 5 bereits vorgestellten *Meier*-Varianten erkannt. *Major* hingegen wird mit unterschiedlichem Soundex-Code verschlüsselt.

Name	Soundex-Code
Maier	M600
Mayer	M600
Meier	M600
Meyer	M600
aber:	
Major	M260
Beier	B600

Name	Soundex-Code
Falsch Positiv:	
Spears	S162
Superzicke	S162
Falsch Negativ:	
Fischer	F260
Vischer	V260
Moskowitz	M232
Moskovitz	M213

Beispiel 2.1: Namen codiert mit Soundex

Vorteile

- Ähnlich klingende Namen aus dem englischen Sprachraum werden mit großer Wahrscheinlichkeit erkannt. Für andere Sprachräume müssen aber erhebliche Änderungen vorgenommen werden.
- Der Algorithmus lässt sich leicht ohne Computerunterstützung auch von Hand durchführen.
- Einige Datenbanken, wie z. B. Oracle oder MySQL unterstützen Soundex direkt über proprietäre Erweiterungen der Abfragesprache. Siehe hierzu auch Kapitel 3.1.
- Codes können im Voraus berechnet und mit in der Datenbank abgespeichert werden. Die Suche ist dann ein einfacher Vergleich der Codes. Dadurch hohe Effizienz.
- Implementierungen für zahlreiche Programmiersprachen existieren. In einigen Sprachen (wie z. B. PHP ab Version 4) ist die Soundex-Codierung direkt im Sprachumfang enthalten.

Nachteile

- Soundex ist sehr auf den englischsprachigen Raum festgelegt und berücksichtigt weder anderssprachige Aussprachen, noch verschiedene Transkriptionen von Namen aus anderen Schriftzeichen in lateinische Buchstaben.
- Abweichungen an der ersten Stelle eines Wortes führen immer zu unterschiedlichen Codes.

– Bei Namenszusätzen wie *Von*, *Van*, *De*, *Zu*, u. Ä. ist nicht eindeutig klar, ob der Zusatz mit codiert wurde oder nicht. Es entstehen dadurch aber ganz unterschiedliche Codes.
– Das Verfahren ist relativ grob. Bei Namen mit vielen Vokalen können z. B. auch sehr unterschiedlich klingende Namen wie *Spears* und *Superzicke* den selben Soundex-Code erhalten.
– Nicht robust gegen Tippfehler wie z. B. Zeichendreher.

2.1.2 Daitch-Mokotoff-Soundex

Randy Daitch und Gary Mokotoff überarbeiteten und erweiterten das ursprüngliche Soundex-Verfahren von Russel. Sie stellten fest, dass gerade yiddische oder slawische Namen mit dem ursprünglichen Verfahren keine guten Ergebnisse liefern, da der Russel-Soundex nur für englische Namen erstellt wurde (vgl. [Mok97]).

Der Daitch-Mokotoff-Soundex-Algorithmus ist wesentlich komplexer als der traditionelle, gewinnt dadurch aber deutlich an Genauigkeit, vor allem für slawische und yiddische Namen. Er wird hier exemplarisch vorgestellt, um mögliche Ansätze zur Verbesserung der Ergebnisse vorzustellen.

Änderungen des Daitch-Mokotoff-Soundex zur ursprünglichen Version von Russel:

1. Das Codierschema enthält deutlich mehr Regeln. Für den interessierten Leser ist das komplette Schema im Anhang A abgebildet.
2. Der erste Buchstaben des Namens wird codiert nach den selben Regeln wie Buchstaben innerhalb des Wortes. Eine Ausnahme bilden Vokale, die am Wortanfang mit 0 codiert werden.
3. Buchstabenkombinationen, die nur für einen Laut stehen, wie z. B. *ts* oder *tz*, werden auch nur mit einer Ziffer codiert.
4. Es werden Codes mit einer Länge von 6 Zeichen erzeugt, anstatt nur 4.
5. Falls Zeichenkombinationen zwei verschiedene Laute repräsentieren können (z. B. kann *ch* wie in *Christian* oder in *Charles* gesprochen werden), müssen beide Laute codiert werden.

Beispiele für Daitch-Mokotoff Soundex Codierungen

Beispiel 2.2 zeigt, dass die Änderungen am Algorithmus eine deutlich bessere Anpassung für osteuropäische Namen bietet. Es zeigt aber auch in der rechten Tabelle, dass für einen Namen mehrere Codes existieren können, was die Komplexität steigert. Um z. B.

zu vergleichen, ob *Spears* und *Superzicke* phonetisch ähnlich sind, müssen 8 Vergleiche ausgeführt werden.

Name	DM-Code	Soundex
Auerbach	097500	A612
Ohrbach	097500	O612
Ceniow	467000	C500
Tsenyuv	467000	T251
Holubica	587400	H412
Golubitsa	587400	G413

Name	DM-Code	Soundex
Spears	479400	S162
	474000	
Superzicke	474500	S162
	479459	
	474450	
	479445	

Beispiel 2.2: Namen codiert nach Daitch-Mokotoff im Vergleich zu Soundex

Vorteile

- Gute Anpassung an osteuropäische Namen. Das Verfahren findet daher auch hauptsächlich Verwendung in der Ahnenforschung von slawischen und jüdischen Personenstandsdatenbanken.
- Bessere Genauigkeit bei längeren Namen.
- Codes können im Voraus berechnet und mit in der Datenbank abgespeichert werden. Die Suche ist dann ein einfacher Vergleich der Codes. Dadurch hohe Effizienz.

Nachteile

- Für ein Wort können mehrere Codes entstehen, was die Komplexität bei Vergleichen erhöht.
- Komplexe Abwandlung des Russel-Soundex – eine Berechnung ohne Computer ist vergleichsweise aufwendig.
- Für Namen aus anderen Sprachräumen ist das Verfahren nicht optimal.

2.1.3 Kölner Phonetik

Eine weitere Abwandlung des klassischen Soundex-Algorithmus, die im Rahmen dieser Arbeit betrachtet werden soll, ist die Kölner Phonetik. Sie wurde als früher Ansatz zur Anpassung des Russel-Soundex an die deutsche Aussprache von H.-J. Postel in

[Pos69] veröffentlicht. Anders als bei Soundex gibt es keine Längenbeschränkung für die Codes.

Codierungsregeln der Kölner Phonetik:

1. Von links nach rechts wird buchstabenweise nach dem Codierschema in Tabelle 2.2 codiert.
2. Danach werden alle mehrfachen Codes eliminiert.
3. Eine „0“ an der ersten Stelle bleibt erhalten. Alle anderen „0“ innerhalb des Codes werden entfernt.

Tabelle 2.2: Codierschema der Kölner Phonetik

Buchstabe	**Kontext**	**Code**
A, E, I, J, O, U, Y, *Ä, Ö, Ü*		0
H		-
B		1
P	nicht vor H	1
D, T	nicht vor C, S, Z	2
F, V, W		3
P	vor H	3
G, K, Q		4
C	im Anlaut vor A, H, L, O, Q, R, U, X	4
C	vor A, H, K, O, Q, U, X außer nach S, Z	4
X	nicht nach C, K, Q	48
L		5
M, N		6
R		7
S, Z, *ß*		8
C	nach S, Z	8
C	im Anlaut vor A, H, K, L, O, Q, R, U, X	8
C	nicht vor A, H, K, O, Q, U, X	8
D, T	vor C, S, Z	8
X	nach C, K, Q	8

Beispiele für die Kölner Phonetik Beispiel 2.3 zeigt, dass Anpassungen an deutsche Namen die Genauigkeit erhöhen. So erzeugt im Gegensatz zu Soundex die Kölner Phonetik auch für *Fischer* und *Vischer* den selben Code. Allerdings werden auch hier *Meier* und *Beier* nicht als ähnlich erkannt.

Auch für slawische Namen bringt die Kölner Phonetik einige Vorteile im Vergleich zu Soundex, wie aber das Paar *Holubica* und *Golubitsa* zeigt, ist es jedoch für diese Namen dem Daitch-Mokotoff Soundex unterlegen.

Name	Kölner Phonetik	Soundex	Name	Kölner Phonetik	Soundex
Maier	67	M600	Auerbach	0714	A612
Mayer	67	M600	Ohrbach	0714	O612
Meier	67	M600	Moskowitz	68438	M232
Meyer	67	M600	Moskovitz	68438	M213
Major	67	M260	Ceniow	863	C500
Beier	17	B600	Tsenyuv	863	T251
Fischer	387	F260	Holubica	514	H412
Vischer	387	V260	Golubitsa	4518	G413

Beispiel 2.3: Namen codiert nach Kölner Phonetik im Vergleich zu Soundex

Vorteile

- Gute Anpassung an den deutschen Sprachraum.
- Bessere Genauigkeit bei langen Namen.
- Deckt slawische Namen besser ab als Soundex nach Russel.
- Codes können im Voraus berechnet und mit in der Datenbank abgespeichert werden. Die Suche ist dann ein einfacher Vergleich der Codes. Dadurch hohe Effizienz.
- Einfaches Verfahren.

Nachteile

- Geringe Verbreitung – kommerziellen Lösungen enthalten normalerweise keine Implementierungen.

2.1.4 Weitere Soundex-basierte Algorithmen

Zahlreiche weitere Verbesserungen und Anpassungen von Soundex führten zu einer Vielzahl weiterer phonetischer Verfahren, die alle mehr oder weniger die bereits vorgestellten Prinzipien von Soundex mit leicht veränderten Regeln oder Codetabellen für bestimmte Sprachräume oder Anwendungsfälle zu optimieren versuchen. Im Rahmen dieser Einführung soll nicht näher auf diese Verfahren eingegangen werden. Folgende Verfahren seien jedoch noch kurz erwähnt:

Metaphone enthält etwas ausführlichere Regeln als Soundex und berücksichtigt auch den Kontext eines Zeichens. Die resultierenden Schlüssel bestehen aus Buchstaben (und wenigen Ziffern für bestimmte Laute, z.B. „0“ für das englische „th“) und haben eine variable Länge (vgl. [Wil05, S. 14]). Das Verfahren ist relativ weit verbreitet und Funktionen zur Berechnung des Metaphone-Codes gehören in manchen Programmiersprachen zum Grundumfang, wie z. B. die Funktion metaphone() in PHP ab Version 4. Des weiteren existieren Implementierungen in PL/SQL für Oracle. Siehe hierzu auch Kapitel 3.1.2.

Double Metaphone ist eine auf Metaphone aufbauende Verbesserung, die für einen Namen neben einem Primärcode auch einen alternativen Code erzeugt, der alternative Aussprachemöglichkeiten berücksichtigt, und dadurch bessere Treffer erzielen kann. Double Metaphone basiert auf sehr umfangreichen und kontextbezogenen Regeln und versucht Unregelmäßigkeiten aus vielen (hauptsächlich europäischen) Sprachräumen zu berücksichtigen. Das führt vereinzelt zu über 100 Regeln für nur einen Buchstaben.

2.1.5 Match Rating Approach

Der Match Rating Approach ist ein weiteres phonetisches Verfahren, das 1977 von den Western Airlines entwickelt wurde (vgl. [Arm00, S. 39 f.]). Es verfolgt einen komplett anderen Ansatz als Soundex und soll darum ebenfalls kurz vorgestellt werden.

Der Algorithmus zeichnet sich durch sehr einfache **Codierungsregeln** aus:

1. Alle Vokale (außer am Wortanfang) werden aus dem Namen gelöscht.
2. Alle Doppelkonsonanten werden in einzelne Konsonanten geändert.
3. Falls der resultierende Code länger als 6 Zeichen ist, werden nur die ersten 3 und letzten 3 verwendet.

Dafür sind die **Vergleichsregeln** vergleichsweise umfangreich:

1. Falls sich die beiden zu vergleichenden Codes in der Zeichenlänge um mehr als 3 Zeichen unterscheiden, werden sie ohne weitere Tests als ungleich klassifiziert.
2. Die Zeichenlänge der beiden Codes wird addiert und für die resultierende Summe ein Schwellenwert aus der Tabelle 2.3 ermittelt.
3. Danach werden beide Namen von vorne (links nach rechts) Zeichen für Zeichen verglichen und identische Zeichen aus beiden Namen entfernt.
4. Die verbleibenden Codes werden von hinten (rechts nach links) Zeichen für Zeichen verglichen und identische Zeichen aus beiden Codes entfernt.
5. Von dem längeren verbliebenen Code wird die Zeichenlänge ermittelt und vom Wert 6 subtrahiert. Die resultierende Zahl ist der Ähnlichkeitswert.
6. Wenn der Ähnlichkeitswert größer oder gleich dem in Schritt 2 ermittelten Schwellenwert ist, gelten die zwei verglichenen Namen als ähnlich; andernfalls sind sie verschieden.

Summe der Zeichenlänge	Minimaler Ähnlichkeitswert
bis 4	5
5 bis 7	4
8 bis 11	3
12	2

Tabelle 2.3: Schwellenwerte für den Match Rating Approach

Beispiele für den Match Rating Approach

Beispiel 2.4 zeigt, wie jeweils zwei Namen mit Hilfe des Match Rating Approach auf Ähnlichkeit überprüft wurden.

Name	MRA-Code	Schwellenwert	Ähnlichkeitswert	Treffer
Schmied Schmidt	SCHMD SCHMDT	3	5	✓
Maier Meyer	MR MYR	4	5	✓
Mayer Beier	MYR BR	4	4	✓
Major Beyer	MJR BYR	4	4	✓
Ceniow Tsenyuv	CNW TSNYV	3	1	✗
Holubica Golubitsa	HLBC GLBTS	3	3	✓

Beispiel 2.4: Überprüfung der Ähnlichkeit jeweils zweier Namen nach Match Rating Approach

Vorteile

- Vergleichsweise gute Genauigkeit als phonetisches Verfahren, kann aber auch einfache Vertipper erkennen.

Nachteile

- Kann nicht im Voraus berechnet werden, sondern muss für jeden Vergleich getrennt berechnet werden und ist daher nicht so performant wie die bereits vorgestellten Verfahren.
- Geringe Verbreitung – kommerziellen Lösungen enthalten normalerweise keine Implementierungen.

2.2 Distanzverfahren

Einen komplett anderen Ansatz verfolgen die Distanzverfahren. Hierbei wird die Ähnlichkeit nicht über den Klang eines Namens bestimmt, sondern über die Anzahl der veränderten Zeichen. Je nach Metrik, die für die Distanz verwendet wird, erhält man einen Wert für die Ähnlichkeit zweier Namen. In diesem Kapitel werden vier verschiedene Distanzmetriken vorgestellt: Schreibmaschinendistanz, Hamming-Distanz, Levenshtein-Distanz und Jaro-Winkler-Ähnlichkeit.

Um jedoch eine Entscheidung treffen zu können, ob zwei Namen ähnlich sind, wird jeweils noch ein Schwellenwert benötigt, da die Verfahren im Gegensatz zu den bereits vorgestellten nicht nur „ähnlich“ und „nicht ähnlich“ als Ergebnis liefern, sondern einen Distanzwert. Je nach Verfahren beschreibt der Distanzwert die Anzahl der unterschiedlichen Zeichen oder die Anzahl der Änderungen, die nötig wären, den einen Namen in den anderen zu überführen.

Die Länge der Namen sollte in die Bestimmung des Schwellenwerts mit einbezogen werden, denn so sind zum Beispiel 3 notwendige Änderungen in einem Wort mit der Länge von 3 Zeichen ein deutliches Zeichen für keine Ähnlichkeit. 3 notwendige Änderungen in einem Wort von 25 Zeichen Länge hingegen deuten auf große Ähnlichkeit hin.

2.2.1 Schreibmaschinendistanz

Ein sehr einfaches und sehr anschauliches Distanzverfahren ist die Schreibmaschinendistanz, oder auch Tastaturdistanz genannt (siehe [Wik]). Hintergrund dieses Verfahrens ist das Erkennen von Tippfehlern. So ist es wahrscheinlicher, dass man einen Buchstaben fälschlicherweise durch einen anderen ersetzt, der auf einer Tastatur direkt daneben liegt, als mit einem, der weiter von dem korrekten Buchstaben entfernt ist.

Zum Beispiel könnte in einem phonetischen Verfahren die Namen *Scheck* und *Schöck* als sehr ähnlich eingestuft werden. Die Schreibmaschinendistanz ist jedoch 7, was einen versehentlichen Vertipper fast ausschließt.

Andererseits hätte *Apple* und *Wople* nur eine Distanz von 2, was eine Ähnlichkeit nach dieser Metrik schon eher nahe legt.

Der Schreibmaschinendistanz wird nach folgenden Regeln berechnet:

1. Vergleiche beide Namen zeichenweise von rechts nach links.

2. Wenn zwei Zeichen an der jeweils selben Stelle in beiden Namen unterschiedlich sind, berechne, wie viele Tasten dazwischen liegen, und addiere den Wert zu der Gesamtdistanz.

Beispiele für die Tastaturdistanz

Beispiel 2.5 zeigt, wie für jeweils zwei Namen die Distanz berechnet wird.

Name 1:			A		P		P		L		E				
Name 1:			W		O		P		L		E				
Distanz:			1	+	1	+	0	+	0	+	0	=	**2**		
Name 1:	S		C		H		E		C		K				
Name 1:	S		C		H		Ö		C		K				
Distanz:	0	+	0	+	0	+	7	+	0	+	0	=	**7**		
Name 1:			M		A		Y		E		R				
Name 1:			M		E		I		E		R				
Distanz:			0	+	2	+	7	+	0	+	0	=	**9**		

Beispiel 2.5: Tastaturdistanz zwischen jeweils zwei Namen

Vorteile

– Das Verfahren eignet sich hervorragend, um bei versehentlichen Vertippern bei der Eingabe auf der Tastatur trotzdem Ähnlichkeiten festzustellen.

Nachteile

– Wie das *Scheck*/*Schöck*-Beispiel zeigt, kann eine (tatsächlich häufig vorkommende) Vertauschung von nur einem Zeichen einen hohen Distanzwert liefern.
– Eine QUERTY-Tastatur liefert unter Umständen andere Ergebnisse als eine QUERTZ-Tastatur oder gar eine Tastatur mit DVORAK-Layout. Die verwendete Tastatur muss darum bekannt sein, um ein genaues Ergebnis zu erhalten.
– Muss für jeden Vergleich mit jedem Namen neu berechnet werden und ist daher eher aufwendig.

2.2.2 Hamming-Distanz

Die Hamming-Distanz wurde von Richard W. Hamming in [Ham50] eingeführt und ist ein Maß für die Unterschiedlichkeit von Zeichenketten. Hierbei wird die Abweichung in zwei gleich langen Zeichenketten als Distanzwert ausgedrückt.

Die Hamming-Distanz wird nach folgenden Regeln berechnet:

1. Vergleiche beide Namen zeichenweise von rechts nach links.
2. Wenn zwei Zeichen an der jeweils selben Stelle in beiden Namen unterschiedlich sind, erhöhe den Distanzwert um 1.

Das Ergebnis ist also die Anzahl der verschiedenen Zeichen in zwei gleich langen Zeichenketten.

Beispiele für die Berechnung der Hamming-Distanz

Beispiel 2.6 zeigt, wie für jeweils zwei Namen die Distanz berechnet wird.

Name 1:	S		C		H		E		C		K		
Name 1:	S		C		H		Ö		C		K		
Distanz:	0	+	0	+	0	+	1	+	0	+	0	=	**1**
Name 1:			M		A		Y		E		R		
Name 1:			M		E		I		E		R		
Distanz:			0	+	1	+	1	+	0	+	0	=	**2**

Beispiel 2.6: Hamming-Distanz zwischen jeweils zwei Namen

Vorteile

- Verfahren eignet sich gut zu Erkennung von Vertippern.
- Berechnung hängt nicht von einem Tastaturlayout ab und braucht daher keine Informatationen über das verwendete Layout.
- Kann einfach bestimmt werden.

Nachteile

- Kann nur für den Vergleich zweier gleich langer Namen verwendet werden.
- Muss für jedes Paar extra berechnet werden.

2.2.3 Levenshtein-Distanz

Vladimir L. Levenshtein stellt in [Lev65] die sogenannte Levenshtein-Distanz vor, die auch *Edit-Distanz* oder *Editierabstand* genannt wird. Dabei wird ähnlich wie bei der bereits vorgestellten Hamming-Distanz die Anzahl der Änderungen erfasst, die nötig wären, um den ersten Namen in den zweiten zu überführen. Das Verfahren kennt 3 Operationen:

1. Einfügen eines Zeichens
2. Löschen eines Zeichens
3. Vertauschen eines Zeichens

Durch die Operationen „Einfügen“ und „Löschen“ wird damit auch möglich, zwei Namen mit verschiedener Länge zu vergleichen.

Berechnung der Levenshtein-Distanz

Zur Berechnung der Levenshtein-Distanz gibt es verschiedene Algorithmen. Eine anschauliche Methode ist die Berechnung in einer Matrix der Dimension $(s+1, t+1)$ mit Hilfe der folgenden Rekursionsgleichung (wobei s und t die Länge der zu vergleichenden Namen sind):

$$
\begin{aligned}
D_{0,0} &= 0 \\
D_{m,0} &= m \\
D_{0,n} &= n
\end{aligned}
$$

und für alle $1 \leq m \leq s; \quad 1 \leq n \leq t$:

$$
D_{m,n} = \min \begin{cases} D_{m-1,n-1} + x \quad \text{wobei } x = \begin{cases} 1 & \text{falls gleicher Buchstabe} \\ 0 & \text{falls Ersetzung} \end{cases} \\ D_{m,n-1} + 1 \\ D_{m-1,n} + 1 \end{cases}
$$

Die Distanz kann dann aus $D_{m+1,n+1}$ abgelesen werden.

Beispiele für die Berechnung der Levenshtein-Distanz

Für die beiden Namen *Mair* und *Meier* soll in Beispiel 2.7 die Distanz ausgerechnet werden. Zuerst wird eine 6 x 5 -Matrix erstellt und danach mit der bekannten Formel ausgefüllt:

	ε	**M**	**A**	**I**	**R**
ε	⓪	1	2	3	4
M	1	⓪	1	2	3
E	2	1	①	2	3
I	3	2	2	①	2
E	4	3	3	②	2
R	5	4	4	3	②

Beispiel 2.7: Berechnung der Levenshtein-Distanz mit Hilfe einer Matrix

Die Distanz beträgt also 2.

Ich empfehle eine Normierung der Distanz auf Werte zwischen 0 (keine Übereinstimmung) und 1 (identische Namen). Dies kann mit folgender Formel erreicht werden (wobei s und t die Länge der zu vergleichenden Namen sind und D der Distanzwert):

$$D_{\text{normiert}} = \frac{\max(s,t) - D}{\max(s,t)}$$

Vorteile

- Anwendbar bei verschieden langen Namen.
- Implementierungen für viele Programmiersprachen vorhanden.
- Bei D_{normiert} zwischen 0.6 und 0.8 (je nach Anwendungsfall) gute Trefferquote.

Nachteile

- Muss für jedes Paar berechnet werden.
- Berechnung ist komplexer als bei den anderen beiden Distanzverfahren.

Damerau-Levenshtein-Distanz

Bei der Damerau-Levenshtein-Distanz wird die Verdrehung zweier Buchstaben als zusätzliche Operation eingefügt. Während bei der Levenshtein-Distanz eine Verdrehung von zwei Buchstaben zwei Operationen erfordert, nämlich jeweils stellenweise die Ersetzung mit dem korrekten Buchstaben, wird bei dieser Erweiterung nur eine Operation benötigt (vgl. [Dam64]).

2.2.4 Jaro-Winkler-Übereinstimmung

Die Jaro-Winkler-Übereinstimmung nach [Win99] berechnet die Ähnlichkeit zweier Namen. Dafür werden nicht nur die Unterschiede wie bei den anderen vorgestellten Verfahren berücksichtigt, sondern auch die übereinstimmenden Zeichen. Damit sollen Eingabefehler bei der Berechnung der Ähnlichkeit mit berücksichtigt werden. Sie soll hier nur kurz der Vollständigkeit wegen erwähnt werden, da sie in Kapitel 3.1.2 als mögliche Lösung für die unscharfe Suche evaluiert wird.

Die Formel zur Berechnung der Jaro-Winkler-Übereinstimmung $JW(s_1, s_2)$ für zwei Namen s_1 und s_2 lautet:

$$JW(s_1, s_2) = \frac{1}{3} \times \left(\frac{c}{|s_1|} + \frac{c}{|s_2|} + \frac{t}{c} \right)$$

wobei
c die Anzahl der übereinstimmenden Zeichen,
$|s_1|$ die Länge des ersten Namens,
$|s_2|$ die Länge des zweiten Namens und
t die Anzahl der notwendigen Änderungen zum Überführen des einen Namens in den anderen sind.

Das Ergebnis der Berechnung ist ein Wert aus $\mathbb{R}$ zwischen 0 (keine Übereinstimmung) und 1 (identisch).

Beispiele für die Jaro-Winkler-Übereinstimmung

Beispiel 2.8 zeigt die Übereinstimmung jeweils zweier Namen.

Name	Ähnlichkeitswert
Schmied Schmidt	0,94
Maier Meyer	0,76
Mayer Beier	0,60
Ceniow Tsenyuv	0.53
Major Beyer	0,46

Beispiel 2.8: Jaro-Winkler-Übereinstimmung jeweils zweier Namen

Vorteile

- Es werden auch die Übereinstimmungen in die Berechnung mit einbezogen, daher genauere Aussagen über die Ähnlichkeit möglich.
- Es steht ein Maß der Ähnlichkeit zur Verfügung.

Nachteile

- Muss für jedes Paar extra berechnet werden.
- Bei hohem Schwellenwert werden nur wenige gute Treffer gefunden, bei niedrigem jedoch viele schlechte.

2.3 Vergleich der Verfahren

Die verschiedenen Verfahren sollen nun kurz in Tabelle 2.4 miteinander verglichen werden.

Bei der Bewertung wurde berücksichtigt, ob der komplexeste Teil des Verfahrens jeweils im Voraus berechnet werden kann, wie z. B. ein Soundex-Code, der dann auch in einer Datenbank abgelegt werden könnte, oder ob für jedes zu vergleichende Paar eine Berechnung durchgeführt werden muss, wie zum Beispiel bei der Levenshtein-Distanz.

Die Komplexität der Codeberechnungsregel wird in drei Stufen eingeteilt, von 1 (einfach) bis 3 (komplex). Soundex hat z. B. sehr einfache Regeln, daher wurde es mit 1 bewertet. Bei z. B. Double Metaphone hingegen kann es für nur einen Buchstaben über 100 Regeln für unterschiedliche Kontexte geben, daher wurde er mit 3 bewertet.

Bei der Komplexität der Vergleichsregel wurde beurteilt, wie aufwendig der Algorithmus zum Vergleichen ist. Da das bei allen soundexbasierten Verfahren ein einfacher Stringvergleich ist, wurden diese alle mit 1 (einfach) bewertet. Mit 3 (komplex) wurde nur die Levenshtein-Distanz berechnet, da hier die komplexeste Berechnung notwendig ist.

Die Verfahren wurden auch nach der subjektiven Qualität der Treffer bewertet. Hier wurde vor allem rein gefühlsmäßig bewertet, ob möglichst viele subjektiv empfunden ähnliche Treffer gefunden wurden, und möglichst wenige Falsch-Positive.

Neben der generellen Verfügbarkeit der einzelnen Verfahren in verschiedenen Programmiersprachen spielt vor allem auch im Hinblick auf Kapitel 4 das Vorhandenensein einer Implementierung für Apache Lucene (bzw. Apache Solr) eine Rolle. So gibt es Implementierungen von Soundex in vielen Programmiersprachen, für die Kölner Phonetik finden sich jedoch kaum welche.

Darüber hinaus ist für die Auswahl eines Verfahrens wichtig, dass dieses nicht nur speziell auf einen bestimmten Sprachraum festgelegt ist.

Tabelle 2.4: Vergleich der Verfahren

Kriterium	**Soundex**	**Daitch-Mokotoff**	**Kölner Phonetik**	**Metaphone**	**Double Metaphone**	**Match Rating Approach**	**Schreibmaschinendistanz**	**Hamming-Distanz**	**Levenshtein Distanz**	**Jaro-Winkler Ähnlichkeit**
Sprachraum	eng.	slaw..	deu.	(eng.)	(eng.)	–	–	–	–	–
Kann im Voraus berechnet werden	✓	✓	✓	✓	✓					
Muss für jedes Paar separat berechnet werden						✓	✓	✓	✓	✓
Codeberechnungsregel (1=einfach, 3=komplex)	1	2	1	2	3	1	–	–	–	–
Vergleichsregel (1=einfach, 3=komplex)	1	1	1	1	1	2	2	2	3	2
Subjektive Qualität der Treffer (1=gut, 5=schlecht)	3	2	2	2	1	1	2	2	1	2
Verfügbarkeit (1=gut, 3=schlecht)	1	2	3	1	1	3	3	1	1	2
In Lucene/Solr vorhanden	✓	✗	✗	✓	✓	✗	✗	✗	✓	✗

2.4 Apache Lucene

Apache Lucene[1] ist eine hochperformante und vielseitige Java-Implementierung[2] einer Volltext-Suchmaschine (siehe dazu auch [HG05]), die bereits in zahlreichen großen Projekten eingesetzt wird, wie z. B. auf den Webseiten Wikipedia[3], Monster[4] und LinkedIn[5] oder auch in Desktop-Anwendungen wie Eclipse[6] für die Suche in der Hilfe. Lucene beschränkt sich dabei auf die Erstellung des Indexes sowie die Suche in diesem Index. Dabei ist Lucene sehr modular aufgebaut und bietet oft mehrere unterschiedliche Implementierungen der einzelnen Komponenten, die je nach Anwendungsfall dann verschieden kombiniert werden können.

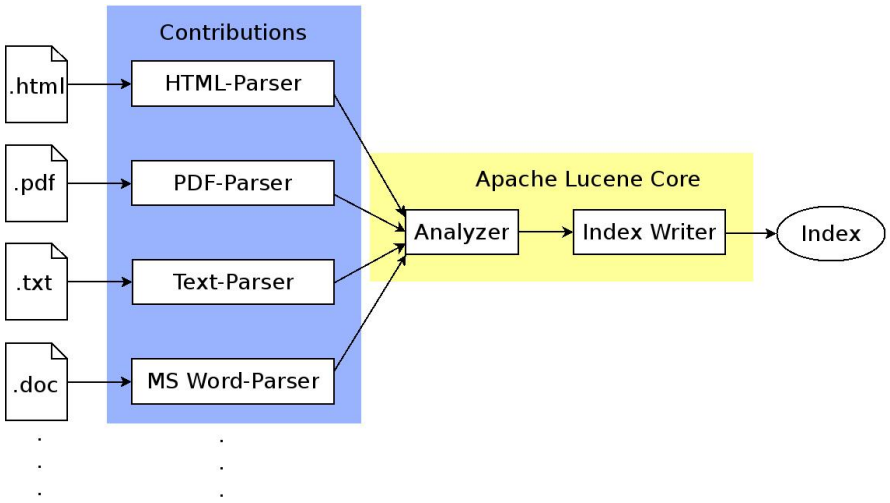
Lucene ist eigentlich auf die Indizierung von Dokumenten ausgelegt. Zusätzlich kann zu jedem Dokument eine beliebige Anzahl von zusätzlichen Daten (z. B. Meta-Daten) in benannten Feldern mit abgelegt werden. Für jedes Feld kann dann entschieden werden, in welcher Form es weiterverarbeitet werden soll, z. B. ob es in einzelne Tokens zerlegt werden soll und ob die Inhalte des Feldes auch gespeichert werden sollen oder nicht.

Das Speichern des Inhaltes eines Feldes hat den Vorteil, dass der komplette Inhalt des Feldes eines Treffers von Lucene auch zurück gegeben wird und so in der Ausgabe einer Suchmaschine direkt angezeigt werden kann, ohne das Dokument öffnen zu müssen. Dem gegenüber steht aber ein deutlich höherer Speicherplatzbedarf im Index.

Zusätzlich zum offiziellen Funktionsumfang gibt es eine ganze Reihe von Open-Source-Erweiterungen, die ebenfalls die Funktionalität von Lucene deutlich erweitern. Des weiteren kann Lucene durch eigene Klassen erweitert und angepasst werden.

[1] http://lucene.apache.org/
[2] Das Hauptprojekt ist die Java-Implementierung. Es gibt jedoch auch Implementierungen für viele weitere Programmiersprachen, wie z. B. .NET, C oder Python
[3] http://www.wikipedia.org/
[4] http://jobsearch.monster.com/
[5] http://www.linkedin.com/
[6] http://www.eclipse.org/

Abbildung 2.1 Schema der Indexerstellung mit Lucene



2.4.1 Erstellen eines Index

Das Erstellen eines Index läuft bei Lucene aus Sicht des Anwendungsprogrammierers in zwei Schritten ab, die vereinfacht in Abbildung 2.1 dargestellt sind:

Im ersten Schritt muss das zu indizierende Dokument geparst werden. Es gibt jedoch bereits verschiedene Parser für gebräuchliche Dokument-Formate, wie z. B. HTML, DOC, PDF oder XML, die verwendet werden können. Das Implementieren eigener Parser ist sehr einfach, damit können beliebige Dokumenttypen indiziert werden.

Im zweiten Schritt werden die Dokumente in den Index aufgenommen. Ein Lucene-Dokument kann dabei noch weitere Felder haben, die Meta-Informationen aufnehmen können.

Der Inhalt des Dokuments wird dabei von einem Analyzer in Wörter zerlegt. Für verschiedene Sprachen gibt es jeweils einen unterschiedlichen Analyzer. Darüber hinaus gibt es auch Stemmer mit Implementierungen für einige Sprachen.

Die offizielle Dokumentation empfiehlt, dass für die Erstellung eines Indexes keine verschiedenen Analyzer gemischt werden sollen, ebenso wie für die Suche in einem Index. Hat man sich für einen Analyzer entschieden, muss für jede weitere Operation der selbe Analyzer verwendet werden.

Stemmer und Lemmatizer

Da in vielen Sprachen Worte je nach grammatischer Stellung verschiedene Endungen haben können, ist es oft nötig, von einem übergebenem Wort die Grundform ableiten können. Dabei wird zwischen zwei verschiedenen Möglichkeiten unterschieden: Stemmer und Lemmatizer.

Bei der Indizierung von Dokumenten, aber auch von Produktdaten usw. spielen Stemmer eine wichtige Rolle für die Genauigkeit der Ergebnisse. Für Stammdaten wie Namen oder Adressen wird kein Stemmer oder Lemmatizer benötigt, da die Ergebnisse in diesem Sonderfall eher verschlechtert werden. Sollten die in dieser Arbeit vorgestellten Methoden jedoch auf andere Daten als Stammdaten angewendet werden, dürfen sie nicht vernachlässigt werden und werden daher kurz erklärt.

Stemmer

Die einfachere Form der beiden ist ein Stemmer (von engl. *stem*: Baumstamm, bzw. *to stem from*: von etwas abstammen). Stemmer haben eine Menge von Regeln, die ein Wort in seine Grundform umwandeln können. In der Regel geschieht dies über das Abschneiden von Wortendungen. So kann ein Stemmer nach einfachen Regeln „*houses*“ durch Abschneiden von „es“ in die Grundform „*house*“ überführen. Stemmer eignen sich für die Englische Sprache deutlich besser als für Deutsch, da im Deutschen immer auch der Wortstamm flektiert wird, wie z. B. „*Haus*“ und „*Häuser*“ zeigt (siehe auch [Tom03]).

Regeln eines Stemmers für Englisch könnten unter anderem enthalten:

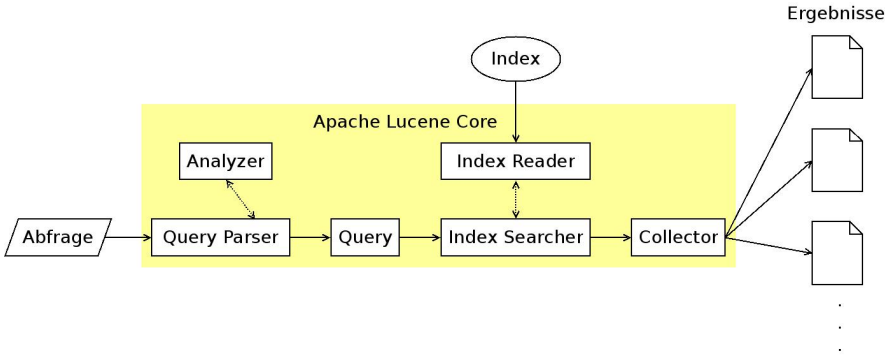
Regel			**Beispiel**		
SSES	→	SS	dresses	→	dress
IES	→	Y	parties	→	party
S	→		books	→	book

Lemmatizer

Ein Lammatizer bestimmt die Grundform nicht durch Heuristiken wie ein Stemmer, sondern benutzt umfangreiche Wörterbücher und Morphemregeln, um Worte korrekter auf die Grundform abzuleiten.

Lemmatizer sind deutlich komplexer und ernstzunehmende Open Source Implementierungen existieren bisher nicht.

Abbildung 2.2 Suche im Lucene-Index



2.4.2 Suche im Index

Die Suche im Index aus Sicht eines Programmierers verläuft ebenfalls in zwei Schritten, die schematisch in Abbildung 2.2 dargestellt werden. Im ersten Schritt wird ein Query-Baum aus `Query`-Objekten erstellt, die dann im zweiten Schritt einem `Searcher` übergeben werden. Die Treffer werden dabei in einem `Collector` ausgewertet.

Es existiert ein `QueryParser`, der einen Suchstring mit Hilfe eines `Analyzer`s in einen passenden Query-Baum überführt. Dies ist eine komfortable Methode zur Erstellung des Suchbaums und wird meistens auch so verwendet. Wie bereits in Kapitel 2.4.1 erwähnt, ist es dabei aber sehr wichtig, den selben Analyzer zum Analysieren des Suchstrings zu verwenden wie beim Schreiben des Index.

Das Erstellen der Query kann aber auch manuell erfolgen, wenn komplexere Query-Bäume benötigt werden, was in Kapitel 4 auch meist notwendig war, weil durch die Kombination verschiedener Verfahren und damit auch verschiedener `Analyzer` im Prototyp der `QueryParser` nicht verwendet werden konnte, da dieser pro Query nur einen `Analyzer` verwenden kann.

Lucene Query Syntax

In diesem Unterkapitel wird die Syntax vorgestellt, die der `QueryParser` von Lucene verwendet, da diese Syntax auch bei einem Anwendungsfall bei der Suche im Prototyp verwendet werden kann.

Eine Abfrage besteht dabei aus einem oder mehreren Termen und Operatoren:

Terme

Einfacher Term ein einfaches Wort. Es wird direkt so eingegeben. Z. B. `buch`

Phrase ist ein Term der aus mehreren Wörtern besteht und genau so auftauchen muss. Eine Phrase wird in Anführungszeichen eingeschlossen. Z. B. `"hallo welt"`

Terme können darüber hinaus noch Meta-Angaben für die Suche enthalten, wie:

Feldnamen Der `QueryParser` verwendet standardmäßig ein Feld des Dokuments zur Suche. Das Standard-Feld wird beim Instantiieren des Query-Parsers angegeben. Soll jedoch in einem anderen Feld gesucht werden, kann der Feldname mit Doppelpunkt vorangestellt werden. Z. B. `titel:apfel` sucht im Feld „titel“ nach dem Wort „Apfel“.

Platzhalter Der Query-Parser erlaubt die Verwendung von Platzhaltern. Dabei steht `?` (das Fragezeichen) für ein beliebiges Zeichen, und `*` (der Stern) für beliebig viele beliebige Zeichen. Platzhalter dürfen nicht am Anfang eines Terms stehen. Z. B. `te?t` findet sowohl „Test“ als auch „Text“.

Unscharfe Suche Lucene unterstützt die unscharfe Suche mit Hilfe der Levenshtein Distanz. Um für einen Simplen Term eine unscharfe Suche durchzuführen, wird ~ (eine Tilde) an das Wort angehängt, und optional direkt daran den Ähnlichkeitswert zwischen 1 und 0, wobei Werte näher zu 1 mehr Ähnlichkeit verlangen. Wird der Ähnlichkeitswert weggelassen, wird standardmäßig 0.5 verwendet. Z. B. `meier~` sucht nach allen Treffern, die mit 0.5 oder mehr eingestuft werden. `meier~0.8` nach Treffern mit mindestens Ähnlichkeit von 0.8.

Kontext-Suche Lucene unterstützt die Suche nach Worten, die nur eine bestimmte Anzahl von Worten auseinander liegen. Dafür wird ~ (die Tilde) auf eine Phrase angewendet. Z. B. `"apfel kuchen"~5` sucht nach Dokumenten, in denen die Worte „Apfel“ und „Kuchen“ nicht weiter als 5 Worte voneinander entfert auftauchen.

Bereichsuche Werte in einem bestimmten Wertebereich können in Verbindung mit dem Schlüsselwort `TO` angegeben werden. Ob die angegebenen Grenzwerte eingeschlossen oder ausgeschlossen sind, wird über eckige (Werte inklusive) und geschwungene (Werte exklusive) Klammern angegeben. Z. B. `preis:[10 TO 100]` findet alle Dokumente, deren Feld „preis“ einen Wert zwischen 10 und 100 hat, jeweils inclusive der Grenzwerte.

Verstärker Einzelnen Termen kann ein stärkeres Gewicht bei der Suche verliehen werden. Dazu wird ^ (Zirkumflex) zusammen mit einem Verstärkunsfaktor („boost factor“) angegeben. Standard ist Wert 1. Negative Werte sind nicht möglich, aber Werte kleiner als 1 können für weniger wichtige Terme verwendet werden. Z. B. `apfel^2 kuchen` liefert Dokumente, die entweder „apfel“ oder „kuchen“ enthalten, aber Dokumente mit „apfel“ werden zweimal höher bewertet.

Operatoren

Mit Hilfe von Operatoren können verschiedene Terme verknüpft werden. Operatoren wie `AND` und `OR` müssen dabei zwingend groß geschrieben werden, um als Operatoren erkannt zu werden.

OR ist ein zweistelliger Operator und gibt an, dass entweder der Term links oder der Term rechts davon im Suchergebnis enthalten sein muss. `OR` ist auch der Standard-Operator: Werden zwei Terme ohne Operator aneinander gereiht, wird eine OR-Verknüpfung angenommen. Z. B. `apfel kuchen` findet Dokumente, die äpfel“ oder „kuchen“ (oder beides) enthalten.

AND ist ebenfalls ein zweistelliger Operator und gibt an, dass sowohl der Term links als auch rechts davon im Suchergebnis enthalten sein muss. Z. B. `apfel AND kuchen` findet Dokumente, die sowohl „apfel“ als auch „kuchen“ beinhalten.

+ ist ein einseitiger Operator und gibt an, dass der folgende Term enthalten sein muss. Z. B. `apfel +kuchen` findet Dokumente, die auf jeden Fall „kuchen“ beinhalten müssen, und „apfel“ beinhalten können.

NOT oder alternativ ! sind zweiseitige Operatoren, die angeben, dass der rechte Term nicht enthalten sein darf. Aus Sicht der Mengenlehre ist es die Differenz. Z. B. `apfel NOT kuchen` findet alle Dokumente, die „apfel“ beinhalten, aber nicht „kuchen“.

- ist ein einstelliger Operator und gibt an, dass der nachfolgende Term in einem Treffer nicht enthalten sein darf. Z. B. `apfel -kuchen` findet alle Dokumente, die „apfel“ beinhalten, aber nicht „kuchen“.

Um verschachtelte logische Ausdrücke beschreiben zu können, dürfen Operatoren auch geklammert werden. Z. B. `(apfel OR kirsch) AND kuchen`.

Um geklammerte Operatoren mit Angabe eines Feldes zu verwenden, wird der Feld-Bezeichner vorangestellt. Z. B. `title:(apfel AND kuchen)`

Beispiele für die Verwendung von Lucene und eine Evaluierung der Leistung finden sich in Kapitel 3.2.

2.4.3 Apache Solr

Apache Solr[7] ist die Implementierung einer Suchmaschine mit Hilfe von Lucene. Unter anderem bietet Solr die komplette Suchmaschinenlogik als Webservice an. Dadurch wird die Anbindung an eigene Anwendungen sehr einfach, da sich die Komplexität der Suche für den Programmierer hauptsächlich auf die Verwendung eines Webservice reduziert.

Dabei bietet Solr verschiedene Erweiterungen, wie z. B. Parser für verschiedene Dokumenttypen. Es stellt aber auch Analyzer für Lucene zur Verfügung, die phonetische Verfahren verwenden. Dadurch wird Solr auch im Zusammenhang dieser Arbeit interessant. Die Verwendung von Solr beschränkt sich jedoch in dieser Ausarbeitung auf eben diese Analyzer.

2.5 Hibernate Search

Hibernate Search[8] ist eine Erweiterung des Persistenz-Frameworks Hibernate[9] um Volltextsuche mit Hilfe von Lucene.

Hibernate bietet ein sehr komfortables Mapping von Objekten zu Datensätzen in einer Datenbank. Über Hibernate Annotations lässt sich der Framework sehr einfach konfigurieren und verwenden.

Da Lucene eigentlich zur Indizierung von Dokumenten entworfen wurde, muss für die Verwendung mit einer Datenbank eine Anpassung vorgenommen werden. Man verwendet in diesem Fall nur die Felder für die Meta-Daten und schreibt die einzelnen Attribute eines Datensatzes in die jeweiligen Felder für Lucene. Dabei ist es sinnvoll, als Feldnamen den selben Bezeichner wie für den Spaltennamen in der Tabelle zu verwenden.

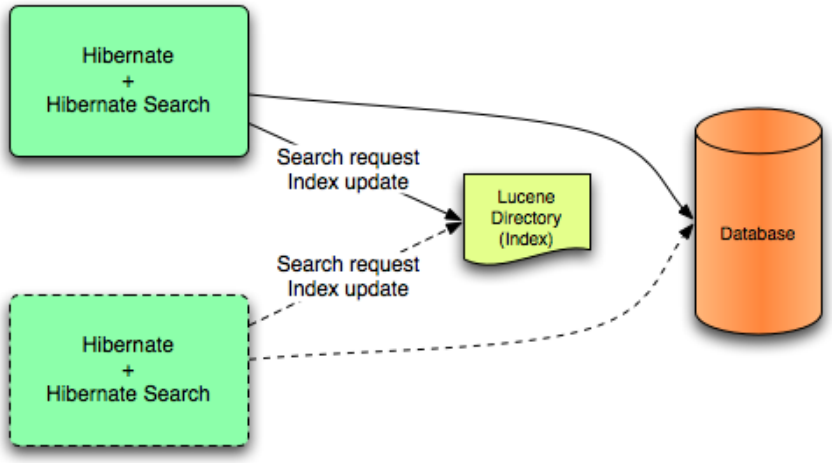
Darüber hinaus müsste der Index manuell aktualisiert werden, wann immer ein Datensatz hinzugefügt, geändert oder gelöscht wurde.

Genau diese Aufgaben nimmt einem Hibernate Search nun ab. Durch wenige einfache Annotations kann das Mapping zwischen Attributen und Lucene-Feldern vorgenommen werden. Zusätzlich liefert die Suche mit Hibernate Search nicht nur die Treffer aus

[7] http://lucene.apache.org/solr/
[8] http://www.hibernate.org/subprojects/search.html
[9] http://www.hibernate.org/

dem Index zurück, sondern über Hibernate gleich echte Objekte (siehe auch Abbildung 2.3).

Abbildung 2.3 Hibernate Search Schema nach [hibb]



Auch das Persistieren und Löschen eines Objekts mit Hilfe des EntityManagers führt automatisch zur Aktualisierung des Index, ohne dass weitere Änderungen an der Programmlogik vorgenommen werden müssen.

Es bietet sich an, für jede Entity einen eigenen Index zu verwenden.

Beispiele für die Verwendung von Hibernate Search finden sich in den Kapiteln 4.2, 4.2 und 4.5.

2.6 BOSCH OpenJava Platform

Für die Erstellung neuer webbasierter Anwendungen bei Bosch wurde die OpenJava Platform standardisiert. Es ist eine Zusammenstellung verschiedener Java-Frameworks und Tools und empfiehlt für die Ausführung einen JBoss Anwendungsserver.

Die aktuelle Version der Platform ist BOSCH OpenJava 1002 und besteht aus folgenden Komponenten:

- JBoss Application Server 5.1.0[10]
- JBoss Seam 2.2.0[11]
- Java SDK 1.6.0.16[12]
- Rich Faces[13]

BOSCH OpenJava beinhaltet außerdem eine IDE und allen benötigten Tools für die Entwicklung:

- Eclipse 3.4.2[14]
- JBoss Tools 3.0.3[15]
- Ant[16]
- ojdbc14[17]
- sowie Tools für das Deployment auf Test- und Qualitätssicherungssystemen

Darüber hinaus gibt es eine Sample-Anwendung, die bereits alle Bosch-spezifischen Anpassungen enthält und sich an die internen Vorgaben hält, wie z. B.

- Anpassung an den BOSCH-Styleguide für Web-Applikationen
- Portalintegration
- Single-Sign-On über das unternehmensweite *WAM*
- Rollenverteilung über *Identity Management*

Für die Entwicklung neuer Applikationen gibt es dazu noch eine build.xml für Ant, die nach Anlegen des Projekts in Eclipse alle Bosch-spezifischen Änderungen in das neue Projekt einpasst.

[10] http://www.jboss.org/jbossas
[11] http://seamframework.org/
[12] http://java.sun.com/javase/
[13] http://www.jboss.org/richfaces
[14] http://www.eclipse.org/
[15] http://www.jboss.org/tools
[16] http://ant.apache.org/
[17] http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/htdocs/jdbc_10201.html

> The nice thing about standards is that you have so many to choose from; furthermore, if you do not like any of them, you can just wait for next year's model.
>
> *– Andrew S. Tanenbaum*

3 Evaluierung verschiedener Lösungsansätze

Um eine unscharfe Suche in die Bosch OpenJava Platform zu integrieren, sind mehrere Ansätze denkbar. In diesem Kapitel sollen verschiedene zur Wahl stehende Möglichkeiten evaluiert werden.

Zum einen wird untersucht, welche Möglichkeiten verschiedene Datenbanksysteme bieten, und ob diese Ansätze in der BOSCH OpenJava Platform verwendet werden könnten. Zum anderen wird die Volltext-Suchmaschine Apache Lucene evaluiert.

Da die BOSCH OpenJava-Platform aus dem Wunsch entstand, eine quell- und lizenzoffene Entwicklungslinie zu schaffen, wurde auf eine Evaluierung proprietärer Bibliotheken, wie z. B. dem Omikron FACT-Finder[18], verzichtet.

Als Datenbasis für die Evaluierung wurden jedes Mal die selben Daten verwendet, um die Ergebnisse vergleichen zu können. Die Datenbasis besteht aus über 115.000 Adressen. Zu weiteren Anmerkungen zu den Adressdaten siehe auch Kapitel 1.

Alle Angaben bezüglich Laufzeit oder Geschwindigkeit in diesem Kapitel beziehen sich auf folgende Rechnerausstattung:

Hardware:

Geräteklasse	Notebook
Prozessor	1,6 GHz Intel Dual Core
Hauptspeicher	2 GB

Sonstiges:

Betriebssystem	Linux, Kernel 2.6.31

[18] http://www.fact-finder.de

3.1 Unscharfe Suche mit Hilfe eines RDBMS

Viele Datenbank-Systeme bieten mittlerweile eine Unterstützung der unscharfen Suche an, entweder über proprietäre Erweiterungen von SQL, oder aber über Stored Procedures. Da für Web-Applicationen im Bosch CI die beiden Systeme MySQL und Oracle angeboten werden, beschränkt sich die Evaluierung auf die beiden Systeme.

3.1.1 MySQL

MySQL kann seit Version 4 unscharfe Suche mit Hilfe von Soundex durchführen. Dazu wird sowohl die Funktion `soundex(`*`name`*`)`, sowie der Operator `SOUNDS LIKE` angeboten. Die Syntax soll durch zwei Beispiele verdeutlicht werden:

```
SELECT *
FROM   kunden
WHERE  name SOUNDS LIKE 'Meier';
```

Oder alternativ

```
SELECT *
FROM   kunden
WHERE  soundex(name) = soundex('Meier');
```

Außerdem kann mit „`SELECT soundex('Meier');`“ der Soundex-Code für einen Namen direkt ausgegeben werden.

Die für diese Evaluierung verwendete Version ist MySQL 5.1.37.

Korrektheit der Berechnung des Soundex-Codes

Zuerst soll überprüft werden, ob die Soundex-Implementierung in MySQL korrekt arbeitet. Dazu sollen die in Kapitel 2.1.1 ermittelten Codes noch einmal von MySQL mit Hilfe von „`select soundex(`*`name`*`)`“ berechnet werden:

Name	Erwartetes Ergebnis	Erhaltenes Ergebnis	Übereinstimmung
Maier	M600	M600	✓
Mayer	M600	M600	✓
Meier	M600	M600	✓
Meyer	M600	M600	✓
Major	M260	M260	✓
Beier	B600	B600	✓
Fischer	F260	F260	✓
Vischer	V260	V260	✓
Moskowitz	M232	M232	✓
Moskovitz	M213	M2132	✗
Superzicke	S162	S162	✓
Scheck	S200	S000	✗
Schweizer	S260	S600	✗

Auffällig ist, dass MySQL teilweise längere Soundex-Codes ausgibt. So erzeugt *Moskowitz* 4 Stellen, *Moskovitz* jedoch 5 Stellen. Die erste Vermutung, dass MySQL die Soundex-Codes nach 4 Stellen einfach nicht abscheidet, wird jedoch durch das Beispiel *Superzicke* widerlegt, das dann nämlich S1622 lauten müsste.

Außerdem fällt auf, dass die Namen *Scheck* und *Schweizer* nicht wie erwartet codiert werden.

Daraus kann geschlossen werden, dass MySQL eine modifizierte Version des Soundex-Algorithmus verwendet. Allerdings zeigen vor allem die letzten beiden Beispiele in der Tabelle, dass die verwendete Version ungenauer als der ursprüngliche Soundex ist.

Suche nach ähnlich klingenden Namen im Datenbestand

Als zweiter Test soll im Datenbestand nach ähnlich klingenden Namen gesucht werden. Dazu wird folgende Abfrage verwendet.

```
SELECT DISTINCT name
FROM   adressen
WHERE  name SOUNDS LIKE 'Name';
```

Für den Namen *Meier* liefert die Anfrage das folgendes Ergebnis:

Gesucht:	Meier
Gefundene Namen:	50
Gute Treffer:	Maier, Meier, Meyer, Mayr, Mayer, Mair, Maeyer, Moier, Mauer, Mayrer, Mahir
Schlechte Treffer:	Maurer, Muhr, Mohr, Mahr, Mauri, Maimer, Mihr, Meurer, Mauerer, Mera, Murrer, Marr, Mahner, Mennemeier, Mueri, Menauer, Memmer, Muehr, Moor, Morawe, Morareo, Mehrer, Marie, Minar, Minner, Morr, Maru, Maehner, Miehr, Murauer, Muhra, Monnier, Maar, Manerer, Moreira, Meer, Meri, Moro, Merry

Die Bewertung der Treffer in gut und schlecht wurde rein subjektiv vorgenommen. Als Beurteilungsgrundlage wurde subjektive phonetische Ähnlichkeit zum gesuchten Namen verwendet.

Es fällt auf, dass die im vorherigen Absatz angenommene Vermutung, die Soundex-Version in MySQL sei von der Genauigkeit eher grob, sich in diesem Beispiel bestätigt hat.

Testweise wurden weitere Abfragen durchgeführt:

Gesucht:	Ohrbach
Gefundene Namen:	2
Gute Treffer:	Ohrbach
Schlechte Treffer:	Orbegozo

Gesucht:	Scheck
Gefundene Namen:	130
Gute Treffer:	Sick, Schoech, Schach, Schack, Schick, Schoch, Scheck, Schaeg, Schoeck, Schuck, Schueck, Schock, Schieck, Seck, Schöck, Schoke
Schlechte Treffer:	Sachs, Schuscha, Saey, Saez, Scho, Schiweck, Seegis, Six, Sagsoez, Saggau, Suske, Suesse, Sigg, Sachse, Schiess, Suck, Skoko, Sack, Sah, Sacoo, Schukies, Sowka, Sass, Scheich, See, Sieg, Sich, Schau, Sasse, Scheja, Sessous, Soco, Sawski, Sause, Saahs, Saks, Siek, Sax, Suess, Schoessow, Schawach, Schuh, Schewe, Sashe, Schieweck, Schug, Sachau, Schikowski, Seeg, Schicho, Schache, Sock, Sieke, Schoeke, Schugk, Schewes, Sueck, Schega, Schieke, Schueschke, Schwake, Su, Sjouke, Sossau, Shoji, Sawy, Soccio, Shea, Schaa, Sykes, Sakowski, Schiegg, Szuecz, Sowa, Sekac, Schawe, Siko, Schiwy, Sachweh, , Soika, Seiz, Soe, Swewczyk, Schweig, Scheu, Seigis, Saucke, Schiwek, Sussieck, Schaack, Soyk, Schauss, Souza, Schuch, Seggewiss, Sois, Saske, Suchy, Sikes, Sewczyk, Sucic, Schwuchow, Schuss, Schaich, Sagzoez, Schuy, Schies, Sacco, Saygi, Szuecs, Schwach, Sues, Skasa, Sy

Beurteilung der phonetischen Suche mit MySQL

Die Suche über 117677 Datensätze geht sehr schnell, die Antwortzeit liegt im Schnitt bei ca. 0,15 Sekunden. Wobei die Suche nach *Tseniov* bei diesem Test am längsten dauerte und ganze 0,27 Sekunden in Anspruch nahm und einen Treffer (*Tsenyuv*) lieferte.

Wie die einfachen Tests gezeigt haben, werden jedoch deutlich mehr Treffer zurückgeliefert als erwartet. Die subjektive Qualität der Treffer ist nicht sehr hoch.

Sollte eine Ähnlichkeitssuche oder Dublettensuche damit realisiert werden, müsste erheblicher weiterer Aufwand unternommen werden, um die gefundenen Treffer noch weiter zu filtern. Die phonetische unscharfe Suche von MySQL kann im Kontext dieser Arbeit eher als eine Vorauswahl für eine Bearbeitung durch genauere Algorithmen angesehen werden, was aber den gesetzten Rahmen überschreitet.

Aufgrund der Ergebnisse dieser Evaluierung, und vor allem da Oracle zum derzeitigen Stand die bevorzugte Datenbank für Projekte im CI ist, wird dieser Ansatz nicht weiter verfolgt.

3.1.2 Oracle

Auch Oracle bietet die phonetische Suche mit Hilfe von Soundex an. Dazu existiert von Hause aus die Funktion `soundex()`. Einen `SOUNDS LIKE` Operator kennt Oracle hingegen nicht. Die Syntax ist identisch mit der Syntax unter MySQL:

```
SELECT *
FROM   kunden
WHERE  soundex(name) = soundex('Meier');
```

Außerdem kann mit „`SELECT soundex('Meier') from dual;`“ der Soundex-Code für einen Namen direkt ausgegeben werden.

Die für diese Evaluierung verwendete Version ist Oracle XE 10g (Version 10.2.0.1-1.0).

Korrektheit der Berechnung des Soundex-Codes

Nachdem MySQL teilweise andere Codes als erwartet zurückgeliefert hat, soll auch unter Oracle überprüft werden, ob die Ausgaben auch der Erwartung entsprechen, bevor weitere Funktionen evaluiert werden.

Name	Erwartetes Ergebnis	Erhaltenes Ergebnis	Übereinstimmung
Maier	M600	M600	✓
Mayer	M600	M600	✓
Meier	M600	M600	✓
Meyer	M600	M600	✓
Major	M260	M260	✓
Beier	B600	B600	✓
Fischer	F260	F260	✓
Vischer	V260	V260	✓
Moskowitz	M232	M232	✓
Moskovitz	M213	M213	✓
Superzicke	S162	S162	✓
Scheck	S200	S200	✓
Schweizer	S260	S260	✓

Bei diesem Test stimmte das erhaltene Ergebnis stets mit dem erwarteten überein. Daraus kann geschlossen werden, dass sich Oracle bei der Implementierung an den Standard gehalten hat und keine Sonderregeln implementiert hat.

Suche nach ähnlich klingenden Namen im Datenbestand

Weitere Tests sollen nun zeigen, wie sich dies auf die Trefferrate auswirkt, wenn im kompletten Datenbestand gesucht wird.

Zur besseren Vergleichbarkeit sollen auch hier wieder die selben Abfragen durchgeführt werden. Der Datenbestand in beiden Datenbanken ist identisch.

Für die Suche wird die folgende Abfrage verwendet:

```
SELECT DISTINCT name
FROM   adressen
WHERE  soundex(name) = soundex('Name')
```

Gesucht:	Meier
Gefundene Namen:	31
Gute Treffer:	Meyer, Meier, Moier, Mahir, Mayer, Mair, Maier, Mayr, Maeyer, Mauer
Schlechte Treffer:	Mauri, Moor, Maar, Merry, Mohr, Marr, Mera, Morr, Mihr, Miehr, Maru, Muhra, Mahr, Moro, Muehr, Marie, Meer, Meri, Muhr, Mueri, Morawe

Diese Abfrage zeigt, dass die schlechten Treffer aus den MySQL-Ergebnissen für *Meier* sich hier fast halbiert haben, die guten Treffer sind jedoch fast identisch, es fehlt nur *Mayrer*, was aber subjektiv betrachtet auch nur gerade noch ein guter Treffer bei MySQL war.

Gesucht:	Ohrbach
Gefundene Namen:	2
Gute Treffer:	Ohrbach
Schlechte Treffer:	Orbegozo

Dieser Test zeigt, dass die Treffer für *Ohrbach* identisch mit denen von MySQL sind.

Gesucht:	Scheck
Gefundene Namen:	93
Gute Treffer:	Schack, Schaeg, Schöck, Scheck, Schick, Schieck, Schiegg, Schoch, Schock, Schoech, Schoeck, Schoeke, Schoke, Schuch, Schuck, Schueck, Schug, Schugk, Suck, Seck, Schach
Schlechte Treffer:	Saahs, Sacco, Sachau, Sachweh, Sack, Sacoo, Saez, Saggau, Saks, Sashe, Saske, Sass, Sasse, Saucke, Sause, Sawski, Sax, Saygi, Schaack, Schache, Schaich, Schauss, Schawach, Schega, Scheich, Scheja, Schewes, Schicho, Schieke, Schies, Schiess, Schieweck, Schiweck, Schiwek, Schoessow, Schuscha, Schuss, Schwach, Schwake, Schweig, Schwuchow, Seeg, Seiz, Shoji, Sich, Sick, Sieg, Siek, Sieke, Sigg, Siko, Six, Sjouke, Skasa, Skoko, Soccio, Sock, Soco, Soika, Sois, Sossau, Souza, Sowka, Soyk, Suchy, Sueck, Sues, Suess, Suesse, Suske, Szuecs, Szuecz

Beurteilung der phonetischen Suche mit Oracle

Die Suche benötigt auch hier durchschnittlich 0,1 Sekunden für eine Antwort, was die Verwendung in einer Web-Anwendung problemlos erlauben würde. Auch die Ergebnisse sind nach subjektivem Empfinden besser als bei MySQL.

Da aber noch immer viele subjektiv schlechte Treffer in den Ergebnissen enthalten sind, ist das Verfahren nicht optimal. Web-Anwendungen, die eine unscharfe Suche mit geringstem Aufwand und eher als Nebenfunktion benötigen, könnten dieses Verfahren eventuell verwenden.

Um die Vorteile von Hibernate (vor allem objektrelationales Mapping) auch aus einer SQL-Query zu nutzen, kann die Methode `addEntity` benutzt werden:

```
session.createSQLQuery(
        "SELECT * FROM adressen WHERE  soundex(name) = soundex('Name')"
    ).addEntity(Customer.class);
```

Wenn die Anwendung aber mehr Ansprüche an die unscharfe Suche stellt, ist dieses Verfahren nur bedingt geeignet.

Erweiterung mit PL/SQL

Für Oracle existieren viele Implementierungen der phonetischer Suche in PL/SQL, die die standardmäßige Funktionalität verbessern sollen. Stellvertretend für eine Vielzahl verschiedener Skripte wird das Paket „Metaphone for Oracle“ von Scott Stephens (siehe [byt]) untersucht.

Nachdem das PL/SQL-Skript importiert wurde, ist die Berechnung des Metaphone-Codes mit der Funktion `meta.phone()` möglich. Daraus ergibt sich die Abfrage:

```
SELECT DISTINCT name
FROM   adressen
WHERE  meta.phone(name) = meta.phone('Name')
```

Stellvertretend wird die Suche für *Scheck* durchgeführt:

Gesucht:	Scheck
Metaphone-Code:	SXK
Gefundene Namen:	26
Gute Treffer:	Schoeck, Schoeke, Schieck, Schieke, Schueck, Schiegg, Schack, Schock, Schaack, Schöck, Schick, Schuck, Scheck, Schug, Schaeg, Schoke
Schlechte Treffer:	Schega, Schweig, Schiweck, Waschka, Waschk, Waschkau, Waschek, Schwake, Schiwek, Schieweck

Das Ergebnis der Suche wurde deutlich besser, viele subjektiv schlechte Treffer wurden unterdrückt. Allerdings hatte die Anfrage eine Antwortzeit von knapp 72 Sekunden. Damit ist dieses Verfahren in dieser Form für eine Web-Anwendung nicht einsetzbar.

Performanceverbesserung durch Vorausberechnung

Zur Verbesserung der Performance wurde die Tabelle `adressen` um die Spalte `metaphone` erweitert, in welcher der zugehörige Metaphone-Code gespeichert wird. Dadurch muss bei einem `SELECT` nicht mehr für jede Zeile der Code extra berechnet werden. Die Abfrage lautet nun für *Scheck*:

```
SELECT DISTINCT name
FROM   adressen
WHERE  metaphone = meta.phone('Scheck')
```

Die Treffer sind wie erwartet identisch, die Antwortzeit hat sich jedoch auf 30,2 Sekunden verkürzt. Dennoch ist dies für eine Web-Anwendung deutlich zu langsam.

Wird der Metaphone-Code für den Namen allerdings im Voraus berechnet, z. B. mit der Abfrage „`SELECT meta.phone('Scheck')`“, kann die Abfrage signifikant beschleunigt werden.

Die Abfrage

```
SELECT DISTINCT name
FROM   adressen
WHERE  metaphone = 'SXK'
```

dauert nur noch 0,02 Sekunden mit identischen Treffern. Mit einem Trigger könnte sichergestellt werden, dass das Feld `metaphone` immer den korrekten Wert enthält. Damit wäre dieses Verfahren für die unscharfe Kundendatensuche in einer Web-Anwendung anwendbar.

Suche in Oracle mit distanzbasierten Verfahren

Zum regulären Lieferumfang von Oracle gehört seit Version 10g Release 2 auch das undokumentierte Paket `UTL_MATCH`, das zwei Algorithmen zur Berechnung der Ähnlichkeit zweier Strings zur Verfügung stellt (vgl. [HM05, S. 18 ff.]): die Levenshtein-Distanz und die Jaro-Winkler-Ähnlichkeit.

Bevor `UTL_MATCH` das erste Mal verwendet werden kann, muss das PL/SQL-Skript `$ORACLE_HOME/rdbms/admin/utlmatch.sql` vom Datenbankadministrator ausgeführt werden.

Bevor eine Suche damit ausgeführt wird, ist es ratsam, einen sinnvollen Schwellenwert für die Distanz zu wählen, ab welchem man einen Namen nicht mehr als ausreichend ähnlich zum gesuchten Namen ansieht. Auch dieses Verfahren soll stellvertretend mit *Scheck* getestet werden. Als Schwellenwert wurde 3 gewählt. Damit ergibt sich die Abfrage:

```
SELECT DISTINCT name, dist
FROM   (
    SELECT name, utl_match.edit_distance(name, 'Scheck') AS dist
    FROM   adressen
```

```
)
WHERE dist < 3
ORDER BY dist
```

Stellvertretend wird die Suche für *Scheck* durchgeführt:

Gesucht:	Scheck
Treffer gesamt:	77
1 × Distanz 0:	Scheck
11 × Distanz 1:	Schack, Schenck, Schenk, Schick, Schieck, Schneck, Schock, Schoeck, Schreck, Schuck, Schueck
65 × Distanz 2:	Schaack, Schach, Schalk, Schank, Schareck, Scheel, Scheer, Schega, Schehr, Scheib, Scheich, Scheid, Scheikl, Schein, Scheit, Scheja, Schelb, Schell, Schelm, Schels, Schemm, Schenke, Schenkl, Schepe, Scherb, Scherf, Scherg, Scherl, Scherm, Scherr, Schett, Scheu, Scheur, Scheve, Schewe, Schickl, Schimeck, Schink, Schiweck, Schlak, Schlick, Schmok, Schmuck, Schmueck, Schnack, Schnick, Schoch, Schöck, Schoech, Schrack, Schrenk, Schrick, Schroeck, Schuch, Schugk, Schunck, Schurk, Schwenk, Seck, Sobeck, Speck, Steck, Stoeck, Streck, Sueck

Obwohl der Wert für jede Zeile gesondert berechnet werden muss, stand das Ergebnis bereits nach 1,7 Sekunden zur Verfügung. Es fällt außerdem auf, dass die Anzahl der Treffer wieder deutlich höher ist, die Treffer aber nicht nach subjektivem Empfinden in gute und schlechte Treffer gruppiert werden müssen, da ein Distanzwert die Ähnlichkeit angibt. Man erhält dadurch eine Gewichtung, die von der Anwendungslogik ausgewertet und berücksichtigt werden kann.

Jaro-Winkler-Übereinstimmung

Auf die Möglichkeit der unscharfen Suche mit Hilfe der Jaro-Winkler-Übereinstimmung (auch aus dem Paket `UTL_MATCH`) soll ebenfalls kurz eingegangen werden. Die Funktion `UTL_MATCH.jaro_winkler_similarity(s1, s2)` liefert die Übereinstimmung nach Jaro-Winkler in Prozent. Als Schwellenwert wählen wir 90:

```
SELECT DISTINCT name, dist
FROM   (
    SELECT name, utl_match.jaro_winkler_similarity(name, 'Scheck') AS dist
    FROM   adressen
```

```
)
WHERE dist > 90
ORDER BY dist DESC
```

Gesucht:	Scheck
Gefundene Namen gesamt:	29
1 × 100% Übereinstimmung:	Scheck
1 × 97% Übereinstimmung:	Schenck
5 × 96% Übereinstimmung:	Schieck, Schneck, Schoeck, Schreck, Schueck
5 × 94% Übereinstimmung:	Schareck, Schimeck, Schiweck, Schmueck, Schroeck
2 × 93% Übereinstimmung:	Schenk, Schennack
15 × 92% Übereinstimmung:	Schack, Schalueck, Schambeck, Scheffczik, Schencking, Schick, Schieckel, Schiedeck, Schiereck, Schieweck, Schock, Schrecker, Schreieck, Schuck, Schwebcke

Das Ergebnis der Abfrage stand nach 1,6 Sekunden zur Verfügung. Es fällt auf, dass die Ergebnisse noch feiner gewichtet sind als bei der Verwendung der Levenshtein-Distanz. Allerdings fällt ebenfalls auf, dass z. B. *Schenck* besser bewertet wird als *Schneck*, und außerdem dass viele nach subjektivem Empfinden gute Treffer aus anderen Abfragen hier nicht auftauchen.

3.2 Apache Lucene

Apache Lucene bietet standardmäßig eine unscharfe Suche mit Hilfe der Levenshtein-Distanz. Um Lucene evaluieren zu können, wurden zwei Java-Klassen geschrieben, von der eine die schon in den anderen Beispielen verwendeten Adressen in den Index schreibt, und eine zweite zur Suche im Index.

Standardmäßig verwendet Lucene bei der unscharfen Suche mit der Levenshtein-Distanz ein $D_{\text{normiert}} \geq 0.5$. Es hat sich aber gezeigt, dass die subjektive Qualität der Treffer bei $D_{\text{normiert}} \geq 0.8$ deutlich besser wird.

Auch hier soll nach *Ohrbach*, *Scheck* und *Meier* gesucht werden.

Gesucht:	Ohrbach (mit $D_{\text{normiert}} \geq 0.8$)
Gefundene Namen:	4
Gute Treffer:	Ohrbach, Mohrbach, Rohrbach, Ohlbach-Nowatzki
Schlechte Treffer:	–

Die Suche dauerte 0,3 Sekunden. *Auerbach* taucht auch hier (wie bei allen anderen bisher evaluierten Verfahren) nicht bei den Treffern auf, dafür drei weitere Namen, die subjektiv sehr gute Treffer sind, aber bei bisher keinem anderen Verfahren gefunden wurden.

Als nächstes werden die Ergebnisse für *Scheck* betrachtet.

Gesucht:	Scheck (mit $D_{\text{normiert}} \geq 0.8$)
Gefundene Namen:	18
Gute Treffer:	Scheck, Schenck, Schoeck, Schueck, Schöck, Schock, Schack, Schreck, Schuck, Schneck, Schieck, Schick, Schenk, Schreck-Engelhardt, Thomas-Schuck, Lueke-Schuck, Schick-Wagner, Paetz gen. Schieck
Schlechte Treffer:	–

Die Suche dauerte 0,4 Sekunden, auch hier sind die Treffer durchweg subjektiv gut. Wie auch schon beim letzten Beispiel aufgefallen ist, werden erstmals auch Doppelnamen bei der Suche berücksichtigt.

Abschließend soll nun noch die Suche nach *Meier* betrachtet werden.

Bei $D_{\text{normiert}} \geq 0.8$ werden zwar einige Treffer ausgegeben, aber neben *Meier* werden nur Doppelnamen mit *Meier* gefunden. Jedoch taucht keine andere Schreibweise des Namens auf.

Bei $D_{\text{normiert}} \geq 0.7$ werden deutlich mehr Treffer ausgegeben, vor allem auch viele Doppelnamen. Diese sollen der Übersichtlichkeit wegen allerdings ignoriert werden. Neben *Meier* taucht nun auch *Maier* und *Meyer* auf. Dazu kommt noch *Beier*, *Meer* und *Meir*. Allerdings fehlt noch immer *Mayer*.

Bei $D_{\text{normiert}} \geq 0.6$ kommen noch einige Namen hinzu, allerdings wird *Mayer* noch immer nicht gefunden.

Bei $D_{\text{normiert}} \geq 0.5$ findet sich auch *Mayer* unter den Suchergebnissen, allerdings auch viele schlechte Treffer wie z. B. *Weber* oder auch *Reiser*.

Beurteilung der unscharfen Suche mit Lucene

Insgesamt betrachtet funktioniert die Suche mit Lucene sehr gut und performant. Es ist für den Einsatz in einer Web-Anwendung gut geeignet. Ein großer Vorteil gegenüber allen anderen getesteten Verfahren ist das Finden auch von Doppelnamen. Allerdings

wird auch deutlich, dass die Qualität der Ergebnisse stark vom verwendeten Schwellenwert D_{normiert} abhängt.

Da Hibernate Search intern ebenfalls Lucene verwendet, kann bei der Verwendung von Hibernate Search das gleiche Ergebnis erwartet werden.

3.3 Ergebnis der Evaluierung

Die Evaluierung der verschiedenen Möglichkeiten hat gezeigt, dass mit Lucene und einem geeigneten Schwellenwert die besten Ergebnisse erzielt werden konnten. Außerdem ist es das einzige der getesteten Verfahren, das auch Doppelnamen berücksichtigt.

Einziger Nachteil von Lucene im Vergleich zu den anderen Verfahren ist die zusätzliche Verwaltung des Index unabhängig von der Datenbank. Eine Herausforderung ist es daher, den Index immer konsistent zum Datenbank-Inhalt zu halten. Im Zusammenspiel mit Hibernate Search sollte dies jedoch möglich sein.

Eine weitere Herausforderung bei der Implementierung des Prototyps wird es sein, die Suchanfrage derart zu verfeinern, dass möglichst viele gute Treffer gefunden werden, aber gleichzeitig so wenig wie möglich schlechte. Aufgrund der Flexibilität von Lucene und den bereits mit den Standard-Optionen guten Treffern wird Lucene als einzige mögliche Bibliothek zur Realisierung der unscharfen Suche in Betracht gezogen und soll für den Prototyp verwendet werden.

Vision without implementation is hallucination.

– Benjamin Franklin

4 Implementierung

Die in den Kapiteln 2 und 3 gewonnenen Erkenntnisse sollen nun für die Erstellung eines Prototyps einer Web-Anwendung eingesetzt werden. In diesem Kapitel werden die Überlegungen, die Herangehensweise und die zu überwindenden Probleme beschrieben, die während der Implementierung aufgetreten sind, sowie natürlich deren Lösungen.

Dieses Kapitel soll gleichzeitig auch Anleitung, Beispiel und Dokumentation für den CI/AFJ sein, wie andere Projekte mit einfachen Mitteln die selbe Funktionalität einbinden können.

Für den Prototyp wird eine Klasse `Customer` verwendet, die folgende Attribute besitzt:

- *id*
- firstName
- lastName
- street
- city

4.1 Vergleich Ist- und Soll-Architektur bei der Suche in Web-Anwendungen im BOSCH OpenJava Framework

In diesem Abschnitt soll ein Überblick über die bisherige Architektur der Suche in Web-Anwendungen im BOSCH OpenJava Framework gegeben werden. Außerdem wird erläutert, wie die Architektur der Suche bei Verwendung von Hibernate Search aussieht.

Ist-Architektur

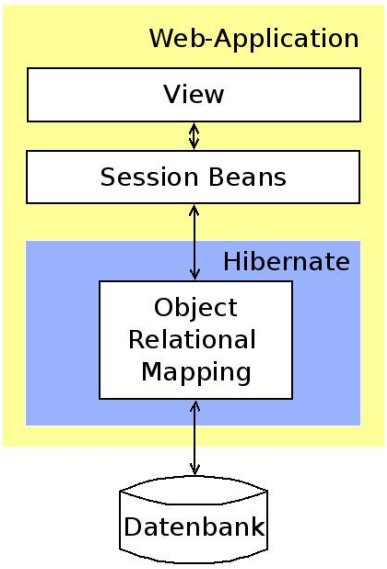
Bisher werden in Anwendungen, die für die BOSCH OpenJava Platform entwickelt werden, für Zugriffe auf Datenbanken Hibernate als Abstraktionsschicht und Framework zur objektrelationalen Abbildung verwendet. Die Architektur ist in Abbildung 4.1 vereinfacht dargestellt.

In den Session-Beans wird von Seam ein `EntityManager` „injected“, über den mit

```
List<Customer> customers = entityManager.createQuery(
    "SELECT customer FROM Customer AS customer "
        + "where lower(name) like lower(?1)")
    .setParameter(1, "Meier%")
    .getResultList();
```

Abfragen in JPQL[19] formuliert werden können und Listen mit Entity-Objekten als Resultat zurückgegeben werden.

Abbildung 4.1 Ist-Architektur für die Suche in Web-Anwendungen bei CI/AFJ mit OpenJava



[19] Java Persistence Query Language. Siehe http://java.sun.com/javaee/5/docs/tutorial/backup/update3/doc/QueryLanguage.html

Soll-Architektur

Beim Einsatz von Hibernate Search wird statt dem `EntityManager` ein `FullTextEntityManager` verwendet, der ebenfalls von Seam „injected“ wird. Die Klasse `FullTextEntityManager` erbt von `EntityManager`, daher kann die bisherige Suchmethode auch weiterhin je nach Anwendungsfall parallel zur Suche mit Hibernate Search verwendet werden.

Bei der Suche mit Hibernate Search wird die Abfrage jedoch nicht als JPQL-Abfrage gestellt, sondern ein `Query`[20]-Objekt (bzw. meistens ein Baum von Query-Objekten) an den `FullTextEntityManager` übergeben:

```
FuzzyQuery luceneQuery = new FuzzyQuery(new Term("name", "meier"), 0.7f);
customerList = (List<Customer>) entityManager.createFullTextQuery(
    luceneQuery, Customer.class)
    .getResultList();
```

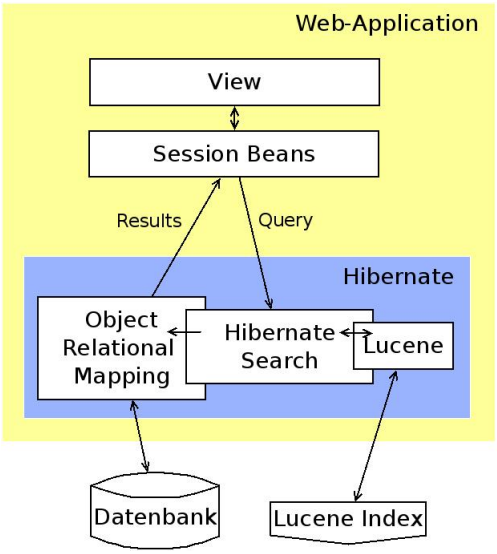
Durch diese Anweisung wird Hibernate Search im Index alle Einträge finden, auf die die Suchkriterien (hier: Name zu 70% identisch mit „meier“) zutreffen. Die IDs der betreffenden Einträge werden dann automatisch von Hibernate Search in eine SQL-Abfrage der Art

```
SELECT *
FROM   Customer
WHERE  id IN (<Liste von den IDs der Treffer>)
```

umgewandelt und ausgeführt. Die Ergebnisse werden dann ebenfalls als Liste von Entity-Objekten zurückgegeben. Einen vereinfachten schematischen Überblick über die Suche mit Hibernate Search zeigt Abbildung 4.2.

[20] Wie eine Query oder ein Query-Baum aufgebaut wird, beschreibt u. a. Kapitel 4.5, sowie [HG05].

Abbildung 4.2 Architektur der Suche in Web-Anwendungen bei CI/AFJ mit OpenJava bei Verwendung von Hibernate Search



4.2 Notwendige Projekteinstellungen zur Nutzung von Hibernate Search im BOSCH OpenJava Framework

Die BOSCH OpenJava Platform bringt die wichtigsten Bibliotheken bereits mit, da Seam Hibernate Search schon integriert hat. Allerdings werden die Bibliotheken beim Anlegen eines neuen Projekts nicht automatisch eingebunden. Daher müssen die folgenden JARs manuell dem Projekt hinzugefügt werden[21]:

- commons-codec.jar
- hibernate-search.jar
- lucene-core.jar

[21] Bei Verwendung der BOSCH OpenJava Platform finden sich die JARs im Verzeichnis `C:\OpenJava-1002\jboss-seam-2.2.0.GA\lib`. Andernfalls können sie zusammen mit Seam von http://seamframework.org/ geladen werden.

Außerdem werden Analyzer und Filter vom Solr-Projekt für die phonetischen Verfahren verwendet. Daher müssen zusätzlich die folgenden JARs[22] mit eingebunden werden:

- apache-solr-common-1.3.0.jar
- apache-solr-core-1.3.0.jar
- apache-solr-solrj-1.3.0.jar

Des weiteren müssen noch zwei zusätzliche Einstellungen in der Datei `persistence.xml` vorgenommen werden (siehe Ausschnitt aus persistence.xml in Listing 4.1).

Listing 4.1 Notwendige Einstellungen in persistence.xml

```
<!-- use a file system based index -->
<property name="hibernate.search.default.directory_provider"
        value="org.hibernate.search.store.FSDirectoryProvider"/>

<!-- directory where the indexes will be stored -->
<property name="hibernate.search.default.indexBase"
        value="index"/>
```

Zum einen muss der zu verwendende Index-Typ angegeben werden. Da der Index nicht in einer Datenbank oder im Hauptspeicher gehalten werden soll, sondern in einer Dateisystem-Hierarchie, wählt man `org.hibernate.search.store.FSDirectoryProvider`.

Als zweites muss dazu noch der Speicherort für den Index angegeben werden.

4.3 Erstellen und Verwalten des Lucene-Index

Damit die Suche im Index mit Lucene überhaupt Resultate finden kann, muss der Index natürlich auch gefüllt werden. Außerdem muss der Inhalt des Index bei jeder Änderung in der Datenbank ebenfalls aktualisiert werden.

In diesem Abschnitt werden zwei Möglichkeiten zum Erstellen und Aktualisieren des Index vorgestellt.

[22] Zum Herunterladen als Teil von Solr unter http://lucene.apache.org/solr/. Es ist darauf zu achten, die Version 1.3 zu verwenden, da die aktuelle Version 1.4 zu Versionskonflikten mit der im BOSCH OpenJava Framework enthaltenen Version von Hibernate führt.

Erweiterungen an Entity-Klassen für die automatische Indexaktualisierung

Hibernate Search bietet einige Annotationen für Entitys, mit denen die komplette Indizierung gesteuert werden kann. Hibernate Search aktualisiert dann automatisch den Index bei jeder regulären Änderung, Neuanlage oder Löschung von Datensätzen.

Wichtig sind drei Annotationen. Sind diese vorhanden, müssen beim Persistieren der Entitys keine weiteren Änderungen in den Entity-Manager-Klassen vorgenommen werden. Ein Minimalbeispiel findet sich im Listing 4.2.

Die Klasse wird mit `@Indexed` annotiert, um zu markieren, dass die Entitys indiziert werden sollen. Zusätzlich wird ein Index-Name angegeben. Hibernate Search erzeugt dann einen Unterordner mit dem selben Namen für den Index in dem Ordner, der bei `hibernate.search.default.indexBase` (siehe Abschnitt 4.2) angegeben wurde. Jede Klasse sollte einen eigenen Indexnamen benutzen, damit die Indizes verschiedener Entitys nicht vermischt werden.

Das Attribut ID der Entity muss mit `@DocumentId` annotiert werden, damit Hibernate Search ein Mapping zwischen Dokumenten im Index und Entitys in der Datenbank herstellen kann. Es zeichnet sozusagen den Primärschlüssel für Hibernate Search aus. Wenn Hibernate Annotations verwendet werden (was bei der BOSCH OpenJava Platform der Fall ist) und ein Attribut mit `@Id` ausgezeichnet ist, kann `@DocumentId` weggelassen werden.

Schließlich werden die zu indizierenden Attribute noch mit `@Field` annotiert. Mit Parametern kann außerdem noch angegeben werden, wie das Feld heißen soll und wie indiziert werden soll: ob der Inhalt in Tokens zerlegt werden soll oder nicht, ob der Inhalt auch im Index gespeichert werden soll, welche Filter und Analyzer für das Feld verwendet werden sollen usw.

Listing 4.2 Minimalbeispiel für Hibernate Search Annotationen (am Beispiel der Entity Foo)

```
@Entity
@Table(name = "foo")
@Indexed(index = "foo")
public class Foo implements Serializable {

    private static final long serialVersionUID = 2318310291911117 98L;

    private Long id;
    private String bar;

    @Id
    @DocumentId
    @GeneratedValue
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @Column(name = "bar")
    @Field(index = Index.TOKENIZED, store = Store.NO),
    public String getBar() {
        return bar;
    }

    public void setBar(String bar) {
        this.bar = bar;
    }
}
```

Manuelle Erzeugung des Lucene-Index

Es gibt einige Fälle, in denen der Index manuell aktualisiert werden muss. Das sind unter anderem:

- Hibernate Search wird nachträglich in ein bereits existierendes System integriert und für die bereits persistierten Objekte gibt es noch keine Einträge im Index.

- Datenbank-Sicherungen wurden eingespielt oder Daten direkt in der Datenbank ohne Benutzung der Anwendung geändert und der Datenbestand passt nicht mehr zum bestehenden Index.

Das Listing 4.3 zeigt beispielhaft, wie innerhalb der Entity-Manager-Klasse `CustomerManager` der Index für die Entity `Customer` neu aufgebaut werden kann. Verwendet wird hierbei nicht der normale EntityManager, sondern wie bei allen Entity-Manager-Klassen, die Hibernate Search benutzen wollen, die Unterklasse FullTextEntityManager, der von Seam automatisch „injected“ wird.

Listing 4.3 Manuelle Indizierung von Objekten (am Beispiel der Entity Customer)

```
@In
FullTextEntityManager entityManager;

public void createIndex() {

    // get all entries from database
    List<Customer> customers = (List<Customer>)
           entityManager.createQuery("select customer from Customer as customer")
                     .getResultList();

    // purge index (in case it already exists)
    entityManager.purgeAll(Customer.class);

    // add each
    for (Customer cust : customers) {
        entityManager.index(cust);
    }
}
```

Da die Laufzeit beim Erstellen des Index bei größeren Tabellen länger als der Timeout des Application-Servers sein kann, bietet es sich an, den Index asynchron erstellen zu lassen. Dazu wird die Methode mit `@Observer(value=ïndexWorker")` annotiert und dann mittels `events.raiseAsynchronousEvent(ïndexWorker");` aufgerufen.

4.4 Vorüberlegungen zur Verbesserung der Suchergebnisse

Der Test von Lucene in Kapitel 3.2 hat gezeigt, dass die unscharfe Suche mit Levenshtein schon sehr gute Ergebnisse liefert. Jedoch hat die Evaluierung ebenfalls gezeigt, dass es auch Schwachstellen gibt. So wurde beispielsweise *Mayer* bei der Suche nach *Meier* nur

gefunden, wenn der Schwellenwert für die Ähnlichkeit relativ gering ist und dadurch auch viele unerwünschte Treffer auftreten. Bei einem phonetischen Verfahren wären die beiden Varianten als ähnlich erkannt worden.

Andererseits kann ein kleiner Tippfehler bei einem phonetischen Verfahren ausreichen, dass (obwohl sich die beiden Namen nur in einem Buchstaben unterscheiden und am Telefon vielleicht sogar sehr ähnlich klingen) keine Ähnlichkeit festgestellt wird: *Müller* und *Füller* werden von fast allen in Kapitel 2.1 vorgestellten Verfahren als unterschiedlich angesehen (Soundex: M460 ≠ F460, Daitch-Mokotoff: 689000 ≠ 789000, Kölner Phonetik: 657 ≠ 357, Metaphone: MLR ≠ FLR), mit Ausnahme des Match Rating Approach (Kapitel 2.1.5), für den es jedoch kaum Implementierungen gibt und der paarweise und daher aufwendiger zu berechnen ist.

Ein distanzbasiertes Verfahren wie Levenshtein hätte jedoch auch noch mit hohem Schwellenwert die Ähnlichkeit entdeckt, stößt dafür aber wieder an Grenzen bei *Schmidt* und *Smith*, die jedoch z. B. von Double Metaphone als ähnlich eingestuft werden.

Diese Überlegungen führten dazu, zwei Verfahren zu kombinieren. Darum soll für diesen Prototyp die distanzbasierte Standard-Fuzzy-Suche von Lucene mit Double Metaphone als phonetisches Verfahren kombiniert werden. Durch die Kombination der beiden Verfahren bleiben die Vorteile der einzelnen Methoden erhalten, aber die jeweiligen Nachteile gleichen sich teilweise aus.

4.5 Notwendige Erweiterungen an Hibernate Search und Lucene zur unscharfen Suche

In diesem Abschnitt soll eine distanzbasierte und eine phonetische Methode für Lucene vereint werden.

Unscharfe Suche mit Levenshtein ist ohne Probleme mit einer `FuzzyQuery` in Lucene bereits verfügbar. Die `FuzzyQuery` funktioniert ohne weitere Vorarbeiten. Auch beim Schreiben des Index muss nichts zusätzlich beachtet werden. Allerdings kann der `QueryParser` nicht direkt verwendet werden, da sonst nur durch Anhängen einer Tilde „~“ an den Suchbegriff unscharf gesucht werden würde. Stattdessen muss die `Query` manuell erstellt werden.

Die Suche mit einem phonetischen Verfahren erfordert allerdings ein paar Änderungen. Zusätzlich zu den normalen Feldern wird jedes Feld noch einmal codiert ebenfalls in

den Index aufgenommen. Um die codierten Inhalte von den uncodierten auseinander halten zu können, werden die codierten in ein eigenes Feld geschrieben.

Der dazu notwendige `Analyzer` muss erst noch definiert werden, was mit Hilfe der Bibliotheken aus dem Solr-Projekt sehr einfach it, siehe auch Listing 4.4. Der Analyzer verwendet einen normalen Tokenizer, damit mehrere Worte in einem Attribut getrennt codiert werden. Zusätzlich wird ein phonetischer Filter hinzugefügt, der hier mit einem `DoubleMetaphone`-Encoder (ebenfalls aus dem Solr-Projekt) die einzelnen Token in phonetische Codes umwandelt, bevor sie in den Index aufgenommen werden.

Listing 4.4 Definition des phonetischen Analyzers

```
@AnalyzerDef(
    name     = "phonetic",
    tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
    filters  = {
        @TokenFilterDef(factory = StandardFilterFactory.class),
        @TokenFilterDef(factory = PhoneticFilterFactory.class, params = {
            @Parameter(name = "encoder", value = "DoubleMetaphone")
        })
})
```

Des weiteren muss bei der Definition der Felder der zusätzliche Encoder angegeben werden. Da der uncodierte und der codierte Attributwert in zwei getrennte Felder geschrieben werden soll, bietet sich an, für das Feld mit den phonetischen Codes den selben Namen mit angehängtem `_pho` zu verwenden. Somit können die beiden Felder jeweils einfach auseinander gehalten werden. Um eine Property in mehrere Felder zu schreiben, muss die `@Fields`-Annotation verwendet werden, wie in Listing 4.5 gezeigt.

Listing 4.5 Ein Objektattribut in mehrere Felder indizieren

```
@Column(name = "city")
@Fields({
    @Field(index = Index.TOKENIZED, store = Store.NO),
    @Field(name = "city_pho", analyzer = @Analyzer(definition = "phonetic"))
})
public String getCity() {
    return city;
}
```

Im Prototyp wurde das für alle Attribute der Klasse `Customer` gemacht. Es werden also auch die Attribute `street` und `city` mit dem phonetischen Analyzer in den Index aufgenommen.

Suche über kombinierte Doppelfelder

Auch bei der Suche über diese Doppelfelder muss die `Query` selber aufgebaut werden. Für jedes Attribut, über das unscharf gesucht werden soll, müssen mindestens zwei Querys (bei dem verwendeten Double Metaphone sogar drei) erzeugt werden, die dann wiederum in einer `BooleanQuery` mit einer ODER-Verknüpfung zusammengefügt werden können. Das heißt, für jeden Term werden intern bis zu 3 Querys verwendet.

Für den Levenshtein-Anteil wird eine `FuzzyQuery` über das Feld mit dem uncodierten Inhalt realisiert. Für den phonetischen Anteil wird eine einfach `TermQuery` über das phonetisch codierte Feld verwendet, aber der Suchstring zuvor ebenfalls in den phonetischen Code umgewandelt. Da Double Metaphone zwei Codes erzeugt, wird für jeden Code je ein TermQuery erzeugt.

Dieses Vorgehen ist für jede unscharfe Suche (und hier wiederum für jedes in der Suche berücksichtigte Feld) notwendig. Darum wurde die Erzeugung dieses Konstrukts in eine statische Methode (siehe Listing 4.6) ausgelagert, die für die eigentlichen Suchen dann beim Zusammenstellen der Querys hilft.

Listing 4.6 Aufbau einer Lucene Query für ein (Doppel-)Feld

```
public static BooleanQuery getCombinedQuery(String field, String value, float boost) {
    BooleanQuery result = new BooleanQuery();
    String metaphonePrimary = encoder.doubleMetaphone(value, false);
    String metaphoneSecondary = encoder.doubleMetaphone(value, true);
    FuzzyQuery fq = new FuzzyQuery(new Term(field, value), 0.7f);
    TermQuery tq1 = new TermQuery(new Term(field + "_pho", metaphonePrimary));
    fq.setBoost(boost);
    tq1.setBoost(boost - 0.1f);
    result.add(fq, Occur.SHOULD);
    result.add(tq1, Occur.SHOULD);
    // check if secondary DoubleMetaphone code is different
    if (! metaphonePrimary.equals(metaphoneSecondary)) {
        TermQuery tq2 = new TermQuery(new Term(field + "_pho", metaphoneSecondary));
        tq2.setBoost(boost - 0.1f);
        result.add(tq2, Occur.SHOULD);
    }
    return result;
}
```

Scores der Treffer erhalten

Sollen von der Anwendung nicht nur die Treffer verarbeitet werden, sondern auch noch der Score für jeden Treffer, kann der Score über eine „Projection“ abgefragt werden. Dazu wird für eine Query die Methode `setProjection` aufgerufen, und die Felder, die im Ergebnis enthalten sein sollen, angegeben.

Der spezielle Wert `FullTextQuery.SCORE` liefert dabei den Score, und `FullTextQuery.THIS` liefert die Entity. Ein Beispiel findet sich in Listing 4.7.

Listing 4.7 Abfrage des Scores zursätzlich zu den Treffern

```
FullTextQuery query = entityManager.createFullTextQuery( luceneQuery, Customer.class)
query.setProjection(FullTextQuery.SCORE, FullTextQuery.THIS);
List results = query.list();
Object[] firstResult = (Object[]) results.get(0);
float score = firstResult[0];
Customer customer = firstResult[1];
```

4.6 Erstellen eines Prototyps

In diesem Kapitel werden die verschiedenen Suchstrategien und Vorgehensweisen für die folgenden Funktionen des Stammdaten-Prototyps beschrieben:

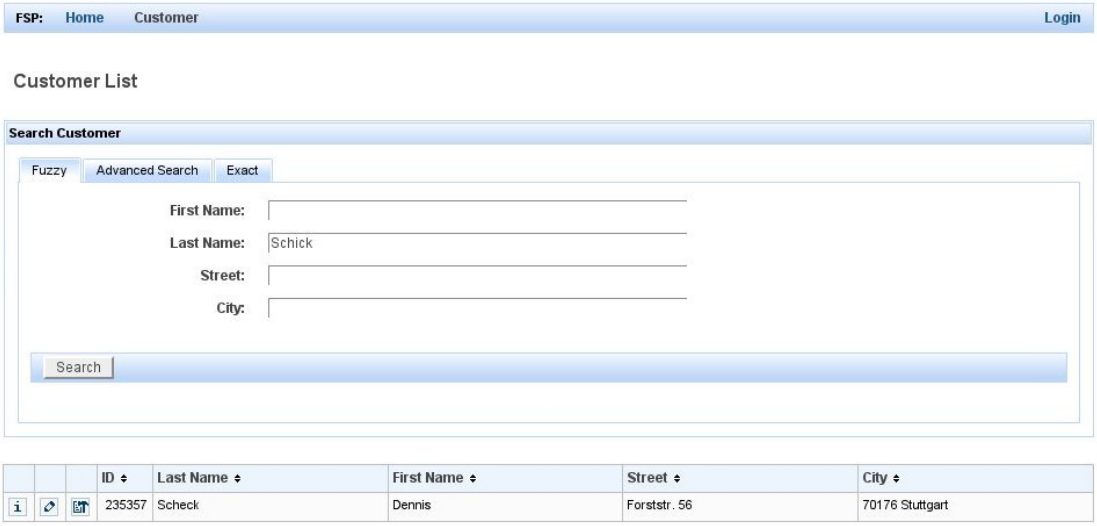
- Unscharfe Suche nach Adressdaten
  - unter Verwendung einer Suchmaske
  - unter Verwendung eines einzigen Suchfeldes
- Dublettenerkennung bei der Neuanlage von Datensätzen
- Zusammenführung mit Daten aus einem anderen System

4.6.1 Unscharfe Suche nach Objekten

Bei neuen Projekten im CI/AFJ tritt immer häufiger die Frage nach unscharfer Suche auf. Dabei gibt es vor allem zwei verschiedene Ausprägungen:

- wie gewohnt mit einer Suchmaske mit Feldern für jedes Attribut
- Suche über alle Attribute durch Eingabe in nur einem Feld, ähnlich wie man es auch bei einer Suchmaschine im Internet kennt

Abbildung 4.3 Attributbezogene Suchmaske für die unscharfe Suche

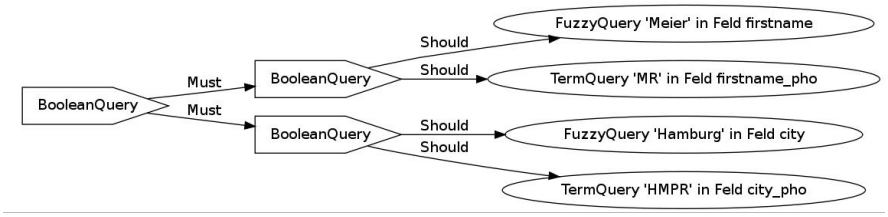


Bei der Suche mit Suchfeldern (siehe Abbildung 4.3) steht eindeutig fest, welcher Term in welchem Feld gesucht werden muss. Eine Query lässt sich darum einfach aufbauen:

1. Für jedes nicht leere Suchfeld wird mit Hilfe von `getCombinedQuery` (aus Listing 4.6) eine Query aufgebaut.
2. Diese Querys werden in einer `BooleanQuery` mit `Occur.MUST` kombiniert.

Der dabei entstehende Query-Baum wird in Abbildung 4.4 veranschaulicht.

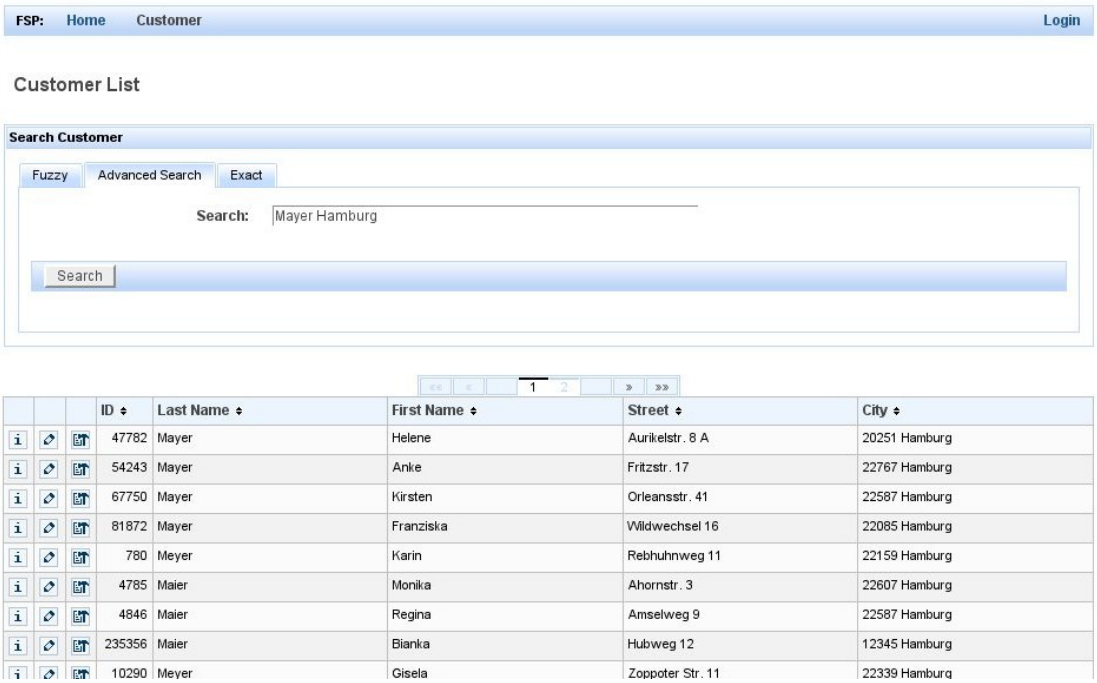
Abbildung 4.4 Baum aus Query-Objekten für die Suche nach Name=„Meier“ und Ort=„Hamburg“



Suche mit nur einem Suchfeld

Soll hingegen mit nur einem Suchfeld wie in Abbildung 4.5 in allen (relevanten) Feldern im Index gesucht werden, muss eine andere Strategie zur Erstellung der Query angewendet werden.

Abbildung 4.5 Suchmaske für die unscharfe Suche über alle Attribute



Die allergrößte Anzahl der Abfragen auf Adressdaten in dieser Maske wird aus einem bis drei Wörtern bestehen, dem Familiennamen, und eventuell dem Vornamen und/oder dem Ort, wie zum Beispiel „*Schmidt*“ oder „*Peter Hinzig Hamburg*“.

Lucene bietet einen `MultiFieldQueryParser`, der einen Suchstring in mehreren Feldern sucht. Doch einerseits wäre damit die Suche über die phonetischen Felder nicht möglich, da der `MultiFieldQueryParser` nur einen Analyzer verwenden kann, und somit unser `phonetic`-Analyzer gar nicht zusätzlich verwendet werden könnte. Ein weiterer Nachteil des `MultiFieldQueryParser` ist, dass bei einer Suche mit Operatoren unerwartete Nebeneffekte auftreten. Zum Beispiel würde eine Suche nach „`Meier AND Hamburg`“ bedeuten, dass mindestens in einem Feld beide Begriffe, also „*Meier*“

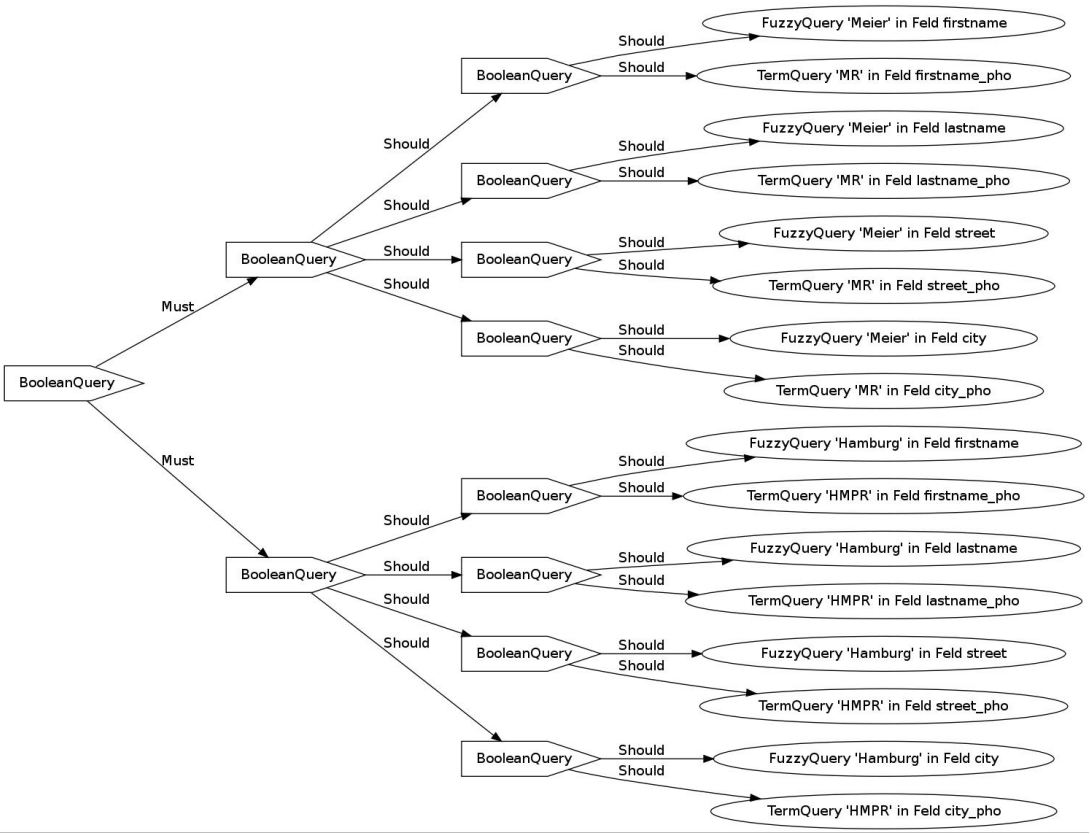
und „*Hamburg*“, vorhanden sein müssen, was mit hoher Wahrscheinlichkeit nicht der Absicht des Suchenden entspricht.

Die Query muss also auch hier manuell aufgebaut werden. Da nicht bekannt ist, welches Wort in welchem Feld vorhanden sein soll, muss ein etwas komplexerer Baum aus `Query`-Objekten aufgebaut werden als bei der Suche mit Suchmaske:

1. Zuerst muss der Suchterm an Leerzeichen in einzelne Tokens aufgespalten werden, da jeder Token in einem anderen Feld vorkommen könnte.
2. Für jeden dieser Tokens wird daraufhin für jedes indizierte Feld mit Hilfe der Methode `getCombinedQuery` (aus Listing 4.6) eine Query erzeugt.
3. Diese Querys werden mit `Occur.SHOULD` in einer `BooleanQuery` kombiniert.
4. Diese Bäume werden wiederum in einer `BooleanQuery` mit `Occur.MUST` kombiniert.

Wie ein Query-Baum für den Suchterm „Meier Hamburg“ aussieht, zeigt Abbildung 4.6.

Abbildung 4.6 Baum aus Query-Objekten für die Suche nach „Meier Hamburg“



Feintuning

Je nach Anwendungsfall und Datenbasis können verschiedene „Schärfegrade“ bei der Suche wünschenswert sein, um mehr oder weniger Treffer zu erhalten.

Dies kann erreicht werden, indem zum einen der minimale Ähnlichkeitswert der `FuzzyQuery` variiert wird; je dichter der Wert bei 1 liegt, desto genauer ist die Suche und desto weniger Treffer werden zurückgeliefert, und je dichter der Wert bei 0 ist, desto unschärfer ist die Suche und desto mehr Treffer werden gefunden.

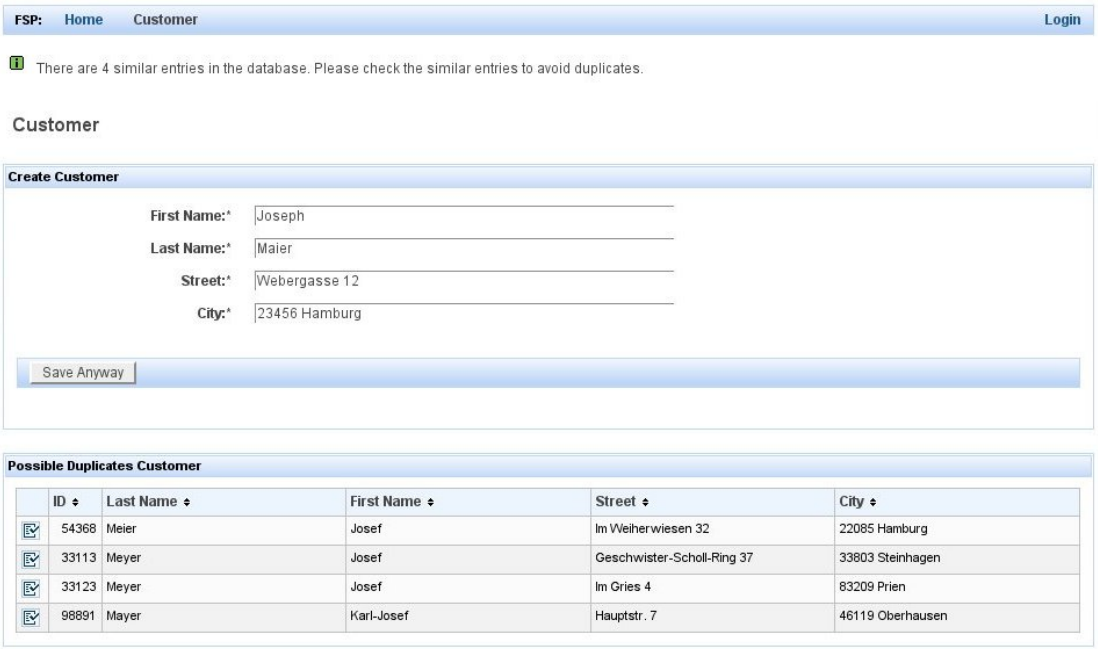
Bei dem phonetischen Anteil kann man den „Schärfegrad“ nicht in diesem Maße steuern. Allerdings hat man hier mit einem Boost-Faktor die Möglichkeit, beim Aufbauen der Query dem phonetischen Feld mehr (Boost-Faktor > 1.0) oder weniger ($0.0 <$

Boost-Faktor < 1.0) Gewicht zu geben. Damit kann man steuern, wie viel Einfluss der phonetische Anteil der Suche auf das Ergebnis hat.

4.6.2 Dublettenlose Neuanlage von Stammdaten

Eine weitere Einsatzmöglichkeit der unscharfen Suche ist die Prüfung auf bereits vorhandene ähnliche Datensätze vor dem Abspeichern einer Neuanlage zur Vermeidung von Dubletten.

Abbildung 4.7 Anzeige möglicher Duplikate beim Anlegen eines neuen Kundenkontos



Zwar könnten Dubletten vermieden werden, wenn der Anwender manuell vor der Neuanlage zuerst eine unscharfe Suche ausführt und überprüft, ob bereits ein identischer Eintrag existiert. Da dies viel Disziplin von den Nutzern erfordert und vor allem bei Zeitdruck unter Umständen vernachlässigt werden würde, wie die Erfahrung mit bereits im Einsatz befindlichen Systemen zeigt, muss die Anwendung diese Funktionalität automatisch abdecken.

Dies wurde folgendermaßen realisiert: Dem Benutzer wird eine gewöhnliche Maske zum Anlegen des Datensatzes angezeigt. Nachdem der Benutzer die Maske ausgefüllt hat, klickt er auf „Speichern“. Findet sich kein ähnlicher Datensatz in der Datenbank, wird ein neuer Datensatz abgespeichert und eine Bestätigung der Speicherung für den Benutzer ausgegeben. Für den Anwender ist der Ablauf soweit also zu 100% identisch mit der Funktionalität, wie sie bisher bei Projekten realisiert wurde. Eine „Umgewöhnung“ der Benutzer zur Vermeidung von Dubletten ist nicht notwendig.

Wird jedoch bei der Prüfung auf ähnliche Datensätze eine potentielle Dublette gefunden, werden die Treffer dem Anwender angezeigt (siehe Abbildung 4.7) und die eingegebenen Daten noch nicht in die Datenbank persistiert. Der Benutzer kann dann selbst entscheiden, ob sein neuer Eintrag in der Liste der angezeigten ähnlichen Einträge vorhanden ist, und dann diesen auswählen, oder aber trotzdem seinen Datensatz wie eingegeben persistieren.

Es wurde versucht, auch Umzüge von Personen zu berücksichtigen, soweit dies möglich ist. Dazu wird zuerst nur nach Einträgen mit ähnlichem Vor- und Nachnamen gesucht, Straße und Ort werden dabei vorerst nicht mit in die Abfrage integriert. Dadurch soll möglich werden, dass bei Personen mit seltenem Namen eventuell ein bereits existierendes Kundenkonto gefunden wird, auch wenn sich die Anschrift geändert hat.

Da bei über 20 Treffern zum einen davon ausgegangen werden kann, dass es sich um einen gebräuchlichen Namen handelt und man weltweit gar nicht mit Sicherheit ein bereits existierendes Konto des selben Kunden identifizieren kann, und zum anderen bei mehr als 20 Einträgen auch die Übersichtlichkeit verloren geht, wird die Suche in diesem Fall unter Einbeziehung des Ortes wiederholt und die Treffer dann angezeigt.

4.6.3 Zusammenführung mit redundanten Daten aus anderen Systemen

Ebenfalls eine häufige Anforderung bei neuen Projekten ist der Abgleich mit anderen Stammdatensystemen. Unternehmensweit existieren viele gewachsene Systeme mit teilweise unüberschaubar vielen Einträgen in den zugehörigen Datenbanken, die oft auch redundante Informationen enthalten, die sich jedoch nicht über Primär- und Fremdschlüssel abgleichen lassen. Der Abgleich muss daher ohne Schlüssel, also über Vergleich der einzelnen Attribute erfolgen.

Da eine scharfe Suche auch hier an Grenzen stößt, wenn leicht abweichende Schreibweisen gewählt wurden, kann auch hier die unscharfe Suche weiterhelfen.

Anders als bei der unscharfen Suche nach einem Datensatz durch einen Benutzer wie in Kapitel 4.6.1, oder nach einer Dublette wie in Kapitel 4.6.2, wo man die Suche eher unschärfer wünscht und eher mehr Treffer bekommen möchte, muss die Suche für diesen Fall genauer sein. Darum werden auf jeden Fall alle in beiden Datenbeständen vorkommenden Attribute als Suchkriterien verwendet.

Bei dem im Prototyp realisierten Beispiel wird für einen `Customer` (als anwendungseigene Entität) aus einer zweiten Tabelle `fsp_client` ein passendes `Client`-Objekt gesucht.

Für den Prototyp liegt die Tabelle im selben Schema, soll aber eine Tabelle aus einer fremden Datenbank repräsentieren. Je nach Datenbank, in der die fremden Daten vorliegen, muss eine unterschiedliche Anbindung realisiert werden[23].

Das Verfahren läuft nach den folgenden Schritten ab:

1. Unscharfe Suche über `Client`, in der alle relevanten Attribute des `Customer` berücksichtigt werden.
2. Gibt es keinen Treffer, konnte kein passender `Client` gefunden werden. Andernfalls werden mit scharfem String-Vergleich für jeden Treffer noch einmal alle Attribute verglichen. Sind sie identisch, kann davon ausgegangen werden, den korrekten `Client` gefunden zu haben.
3. Stimmen im scharfen Vergleich die Attribute bei keinem der Treffer überein, wird die Trefferliste dem Benutzer angezeigt (siehe Abbildung 4.8), damit dieser entscheiden kann, ob der richtigen `Client` in der Liste ist, und diesen dann manuell auswählen.

[23] Liegen die Daten in einer anderen Instanz einer Oracle-Datenbank (was für den Bosch CI der Regelfall sein wird), lässt sich eine fremde Tabelle oder ein fremder View über zwei Schritte im aktuellen Schema zugänglich machen: Zuerst muss mit dem Befehl `CREATE DATABASE LINK` eine Verbindung hergestellt werden, dann kann mit `CREATE SYNONYM` ein Alias im eigenen Schema für die fremde Tabelle angelegt werden.

Abbildung 4.8 Anzeige eines potentiellen übereinstimmenden Datensatz

„Regression testing“? What's that? If it compiles, it is good; if it boots up, it is perfect.

– Linus Torvalds

5 Test und Bewertung der Implementierung

In diesem Kapitel soll der Prototyp aus Kapitel 4 getestet und bewertet werden. Als Bewertungskriterien werden sowohl die Suchgeschwindigkeit, als auch die subjektive Qualität der Such-Ergebnisse verwendet. Außerdem wird untersucht, welche Änderungen sich durch die Kombination zweier unscharfer Suchverfahren im Vergleich zu nur einem Verfahren ergeben.

Für die Bewertungen wurde der Prototyp auf einen der OpenJava POC-Server[24] im Rechenzentrum „deployed“. Alle Angaben bezüglich Laufzeit und Geschwindigkeit in diesem Kapitel beziehen sich auf folgende Rechnerausstattung:

Hardware:

Geräteklasse	Server
Prozessor	3 GHz Intel Quad-Core
Hauptspeicher	8 GB

Sonstiges:

Betriebssystem	Linux, 64-Bit-Kernel 2.6.18

Die verwendete Instanz der Oracle-Datenbank liegt auf einem anderen Server.

5.1 Test und Bewertung der Suchgeschwindigkeit

Um die Geschwindigkeit der Suche zu testen, wurde der Prototyp um einige Debug-Ausgaben ergänzt. Dann wurden die Abfragen über den Browser durchgeführt, jeweils als scharfe Suche und als unscharfe Suche über alle Attribute mit nur einem Suchfeld (siehe 4.5), da hier der Query-Baum komplexer wird als bei der unscharfen Suche mit Suchmaske. Im Anschluss wurde der Serverlog analysiert und die Zeiten abgelesen. Ausgewählte Beispiele sind in der Tabelle 5.1 aufgeführt.

[24] „Proof Of Concept“, diese Server dienen zur Evaluierung und Standardisierung von Verfahren im Umfeld der BOSCH OpenJava Platform

Name	Unscharfe Suche			Scharfe Suche	
	Treffer	Antwortzeit		Treffer	Antwortzeit
		Lucene	Datenbank		Datenbank
Ohrbach	56	$<$ 0,1s	$<$ 0,1s	1	$<$ 0,1s
Scheck	454	~ 0,1s	~ 0,2s	16	$<$ 0,1s
Meier	2337	~ 0,1s	~ 1.6s	149	~ 0,2s
Fischer	621	~ 0,1s	~ 0,3s	474	~ 0,3s

Tabelle 5.1: Geschwindigkeitsvergleich von unscharfer und scharfer Suche im Prototyp

Beim Abschicken von Testabfragen über die Web-Anwendung fiel auf, dass die Zeit, bis die Antwort im Browser erscheint, sehr stark variiert. Abfragen, die nur wenige Treffer zurück liefern, wurden in weniger als 1 Sekunde beantwortet. Bei Abfragen, die mehrere Treffer liefern, dauerte dies teilweise deutlich länger.

Die Debug-Einträge im Serverlog zeigen dabei, dass für die Suche im Lucene Index nur ein zu vernachlässigender Bruchteil der Gesamtzeit verwendet wird. Die größte Anteil entsteht beim Warten auf die Ergebnisse von der Datenbank, nachdem die IDs von Lucene bereits ermittelt wurden. Aber auch die Suche in der Datenbank läuft viel schneller ab, wenn man die selben SQL-Abfragen (mit veränderten IDs, um Vorteile durch Caching im Datenbankserver auszuschließen) direkt eingibt. Darum wurde weiter untersucht, warum im Zusammenspiel die Laufzeit viel länger ist als die Summe der Zeiten der Einzelschritte.

Nach eingehender Überprüfung wurde festgestellt, dass die meiste Zeit für die Übertragung der Daten von der Datenbank zum Applikation-Server über das Intranet benötigt wird, z. B. benötigt die Übertragung aller ~117.000 Einträge im Durchschnitt knapp 15 Sekunden. Negativ beeinflusst wurde dieser Effekt zusätzlich, da wegen eines Netzwerkproblems an den Tagen dieser Tests das Intranet generell sehr langsam war, was sich auch in zahlreichen anderen Web-Anwendungen deutlich bemerkbar machte.

Grundsätzliche Aussagen über die Geschwindigkeit lassen sich daher nicht treffen. Durch die Debug-Einträge im Serverlog konnte jedoch gezeigt werden, dass die Wartezeit nicht durch die Suche im Index entsteht, sondern lediglich durch die Datenübertragung im Netzwerk, die durch Netzwerkprobleme zudem langsamer ablief als üblich. Da hiervon jegliche Art der Suche betroffen ist, wird daraus geschlossen, dass die unscharfe Suche mit Hibernate Search sehr performant abläuft und für den Benutzer

zu keiner wahrnehmbaren Verlängerung der Wartezeit im Vergleich mit der bisher eingesetzten scharfen Suche führt.

5.2 Test und Bewertung der Trefferqualität

Um zu Bewerten, wie sich die Kombination eines Distanzverfahrens und eines phonetischen Verfahrens bei der unscharfen Suche mit Hibernate Search auswirkt, soll für die scharfe Suche, die normale Lucene-`FuzzyQuery` (Levenshtein Distanz), Double Metaphone sowie die Kombination der beiden Verfahren jeweils die Anzahl und die subjektiv empfundene Qualität der Treffer in Schulnoten (1 = sehr gut, 6 = ungenügend) angegeben werden. Die Ergebnisse zeigt Tabelle 5.2.

Zum Ermitteln der Ergebnisse wurde der Prototyp jeweils umgeschrieben, damit die Suche nur das angegebene Verfahren verwendet. Da zur Bewertung der Qualität tatsächlich nur über die Namen gesucht werden soll, wurde hier im Index auch nur im Feld `name` gesucht, daher weicht die Anzahl der Treffer auch im Vergleich zu den Tests in Kapitel 5.2 ab, wo der Term jeweils in allen Feldern gesucht wurde.

Bei der scharfen Suche wurde auf die Bewertung der Qualität verzichtet, da sich dieses Verfahren gerade dadurch auszeichnet, dass die Suchergebnisse stets zu 100% mit der Suchanfrage übereinstimmen. Da die Abfrage mit dem `LIKE`-Operator durchgeführt wurde (siehe Kapitel 4.1), werden trotzdem auch Namen und Doppelnamen gefunden, die nicht genau dem Namen entsprechen, die aber mit genau dem Namen beginnen. Dennoch zeigt die Anzahl der Treffer, wie viele Ergebnisse mit dem BOSCH OpenJava Framework ohne die unscharfe Suche zu erwarten gewesen wären.

Die unscharfe Suche mit der Levenshtein-Distanz wurde für diesen Test mit einem Schwellenwert von 0,5 ausgeführt, wie es der Standardeinstellung des `QueryParser`s von Lucene entspricht. Es fällt auf, dass zwar viele Treffer gefunden werden, jedoch wegen des niederen Schwellenwerts auch viele schlechte Treffer enthalten sind, die für den Benutzer die Übersichtlichkeit der Ergebnisliste einschränken. Die Anzahl schlechter Treffer nimmt deutlich ab, wenn der Schwellenwert erhöht wird, allerdings sinkt damit auch die Anzahl der guten Treffer deutlich.

Bei der unscharfen Suche mit Double Metaphone bleiben die Ergebnisse übersichtlicher. Außerdem sind viele gute Treffer im Ergebnis enthalten. Da es aber hier (im Gegensatz zur Levenshtein-Distanz) keine Gewichtung der Treffer in Form eines Scores gibt, und dadurch auch keine Sortierung, kann es vorkommen, dass der gesuchte Name erst am Ende der Trefferliste erscheint. Zum Beispiel hat *Meier* den Double-Metaphone-Code `MR`.

Damit zählen alle Namen als Treffer, die ebenfalls `MR` als Code besitzen, wie *Mohr*, *Maar* oder *Mera*. Zwar kann es bei der unscharfen Suche durchaus gewünscht sein, diese Treffer ebenfalls zu bekommen, allerdings haben sie den selben Score wie der tatsächlich gesuchte *Meier*, und können daher in der Trefferliste auch davor auftreten.

Bei der Kombination der beiden Verfahren (mit einem Schwellenwert für die Levenshtein-Distanz von 0,7) kann man von den Vorteilen beider Verfahren profitieren. Über das phonetische Verfahren wird sichergestellt, dass trotz hohem Schwellenwert beim Distanzverfahren viele gute Treffer enthalten sind. Das Distanzverfahren hingegen stellt sicher, dass die besten Treffer auch am Anfang der Ergebnisliste auftauchen.

Somit wurde gezeigt, dass durch die Kombination der beiden Verfahren bessere Ergebnisse erhalten werden als durch Verwendung nur eines Verfahrens.

	Scharfe Suche		**Levenshtein**		**Metaphone**		**Kombination**	
Name	Tref.	Qual.	Tref.	Qual.	Tref.	Qual.	Tref.	Qual.
Scheck	16	–	153	1,3	149	2,0	149	1,3
Meyer	195	–	1557	2,7	952	2,3	1085	1,7
Ohrbach	1	–	176	3,0	35	2,7	56	2,0
Vischer	1	–	1552	4,0	496	1,5	598	1,3

Tabelle 5.2: Vergleich der Anzahl und Qualität der Treffer

> Wenn die anderen glauben, man ist am Ende, so muss man erst richtig anfangen.
>
> *– Konrad Adenauer*

6 Zusammenfassung und Ausblick

Im Rahmen dieser Diplomarbeit wurde ein Prototyp erstellt, der für die BOSCH OpenJava Platform mit Hilfe von Hibernate Search und unter Verwendung unscharfer Suchmethoden eine Suche in Stammdaten, eine dublettenlose Neuanlage von neuen Einträgen in ein Stammdatensystem sowie einen Abgleich von Stammdaten mit anderen Systemen realisiert.

Dafür wurden verschiedene phonetische und distanzbasierte Methoden für die unscharfe Suche betrachtet und bewertet. Dabei fiel vor allem auf, dass eine für alle Anwendungsfälle optimale Methode nicht existiert und dass jede Methode ihre Schwachstellen hat.

Im zweiten Schritt wurde untersucht, wie diese Methoden eingesetzt werden könnten. Dazu wurden verschiedene Möglichkeiten zur unscharfen Suche in den RDBMS MySQL und Oracle evaluiert, die aber keine zufriedenstellenden Ergebnisse lieferten. Außerdem wurde die Volltext-Suchmaschine Lucene sowie Solr betrachtet und auch einige Tests damit durchgeführt.

Schließlich wurde versucht, die Vorteile eines phonetischen und eines distanzbasierten Verfahrens für die unscharfe Suche zu kombinieren, um gleichzeitig die jeweiligen Nachteile durch die Kombination auszugleichen. Darum wurden von jeder Methode das am besten bewertete Verfahren ausgewählt und miteinander kombiniert. Das Ergebnis war die Verbindung von Double Metaphone und Levenshtein.

Da Lucene sehr performant arbeitet und in Verbindung mit Hibernate Search sehr komfortabel eingesetzt werden kann, wurde im nächsten Schritt untersucht, wie die Kombination von Double Metaphone und Levenshtein mit Hibernate Search umgesetzt werden kann.

Nachdem die Möglichkeiten zur unscharfen Suche dann durch die Vorarbeit zur Verfügung standen, wurde mit der Implementierung des Prototypen begonnen. Drei verschiedene Anwendungsfälle sollten dabei berücksichtigt werden: Suche von Objekten, dublettenlose Neuanlage sowie Abgleich mit anderen Systemen. Je nach Anwendungsfall waren verschiedene Strategien zum Aufbau einer Query notwendig. Da die BOSCH

OpenJava Platform für mich ebenfalls ein unbekanntes Terrain war, musste ich mich auch hier etwas einarbeiten.

Um im Zusammenspiel der beiden Verfahren die besten Resultate zu erhalten, wurde nach Fertigstellung des Prototypen untersucht, wie sich Änderungen an verschiedenen Parametern, wie z. B. dem Schwellenwert für die Levenshtein-Distanz oder von Boost-Faktoren auf verschiedene Felder auswirken.

Danach wurden die Ergebnisse der Arbeit mit der derzeit eingesetzten scharfen Suche, sowie mit den einzelnen Verfahren verglichen. Dabei wurde deutlich, dass die Kombination der beiden Verfahren die Qualität der Treffer verbessert hat.

Der Prototyp ist einsatzfähig und nutzt die unscharfe Suche erfolgreich für verschiedene Anwendungsfälle. Das Verfahren liefert gute Ergebnisse, einem Einsatz in realen Projekten steht nichts im Wege. Da immer häufiger von Kunden die unscharfe Suche für eigene Projekte gewünscht wird und auch die Vermeidung von Dubletten sowie der Abgleich von Daten in fast jedem Projekt eine Rolle spielen, wird empfohlen, dieses Verfahren in die Standardisierung der OpenJava Platform sowie der Sample-Application aufzunehmen.

Ausblick

In dieser Arbeit wurde nur auf die unscharfe Suche von Namen eingegangen. Um eine flexiblere Lösung zur Abdeckung weiterer Anwendungsfälle zu erhalten, könnten die vorgestellten Methoden erweitert werden.

Denkbare Anwendungsfälle wären z. B. unscharfe Suche für Volltext-Dokumente. Hierbei wird man auf andere Probleme stoßen, die für die Zielsetzung dieser Diplomarbeit nicht relevant waren.

In diesem Zusammenhang können verschiedene Stemmer oder Lemmatizer untersucht werden, auf deren Verwendung bei Namen gänzlich verzichtet werden kann. Auch die Verwendung von Synonymlisten kann hier zu erheblichen Verbesserungen führen.

Außerdem wurde im Prototyp nur nach Strings gesucht. Möchte man auch andere Datentypen indizieren und für die Suche verwenden, werden die Annotationen `@FieldBridge` oder `@DateBridge` benötigt. Aber auch die Auswahl eines anderen Analyzers oder andere Strategien zum Aufbau eines Query-Baums können notwendig werden.

Bisher wurde nur eine Entität berücksichtigt. Möchte man Objekte indizieren, die Pointer auf andere Objekte besitzen, können die Annotation `@IndexedEmbedded` oder `@ContainedIn` benötigt werden. Auch hier werden je nach Anwendungsfall unter Umständen andere Strategien zum Aufbau des Query-Baums notwendig.

Danksagung

Ich möchte mich herzlich bei Frau Dr. Elke Schweizer und Herrn Dipl.-Inf. Alexander Moosbrugger für die kompetente und freundliche Betreuung während der Arbeit und für die konstruktiven Verbesserungsvorschläge und Anregungen bedanken.

Ebenso bei der Abteilung CI/AFJ der Robert Bosch GmbH für die freundliche Aufnahme während der 6 Monate und das große Interesse an den Ergebnissen dieser Arbeit.

Besonders herzlicher Dank gilt meiner Mutter, die mit ihrer Unterstützung mein Studium überhaupt erst ermöglicht hat.

A Daitch-Mokotoff Soundex Kodier-Schema

Das Kodier-Schema zum Daitch-Mokotoff-Verfahren aus Kapitel 2.1.2.

Tabelle A.1: Daitch-Mokotoff Soundex Kodier-Schema

Buchstabe	**alternative Schreibweise**	**Wortanfang**	**vor Vokal**	**sonst**
AI	AJ, AY	0	1	–
AU		0	7	–
A		0	–	–
B		7	7	7
CHS		5	54	54
CH	*siehe KH (5) + TCH (4)*			
CK	*siehe K (5) + TSK (45)*			
CZ	CS, CSZ, CZS	4	4	4
C	*siehe K (5) + TZ (4)*			
DRZ	DRS	4	4	4
DS	DSH, DSZ	4	4	4
DZ	DZH, DZS	4	4	4
D	DT	3	3	3
EI	EJ, EY	0	1	–
EU		1	1	–
E		0	–	–
FB		7	7	7
F		7	7	7
G		5	5	5
H		5	5	–
IA	IE, IO, IU	1	–	–
I		0	–	–

weiter auf der nächsten Seite…

Tabelle A.1 – Fortsetzung

Buchstabe	**alt. Schreibweise**	**Wortanfang**	**vor Vokal**	**sonst**
J	*siehe Y (1) + DZH (4)*			
KS		5	54	54
KH		5	5	5
K		5	5	5
L		8	8	8
MN			66	66
M		6	6	6
NM			66	66
N		6	6	6
OI	OJ, OY	0	1	–
O		0	–	–
P	PF, PH	7	7	7
Q		5	5	5
RZ, RS	RTZ (94) + ZH (4)			
R		9	9	9
SCHTSCH	SCHTSH, SCHTCH	2	4	4
SCH		4	4	4
SHTCH	SHCH, SHTSH	2	4	4
SHT	SCHT, SCHD	2	43	43
SH		4	4	4
STCH	STSCH, SC	2	4	4
STRZ	STRS, STSH	2	4	4
ST		2	43	43
SZSZ	SZCS	2	4	4
SZT	SHD, SZD, SD	2	43	43
SZ		4	4	4
S		4	4	4
TCH	TTCH, TTSCH	4	4	4
TH		3	3	3
TRZ	TRS	4	4	4
TSCH	TSH	4	4	4
TS	TTS, TTSZ, TC	4	4	4

weiter auf der nächsten Seite…

Tabelle A.1 – Fortsetzung

Buchstabe	**alt. Schreibweise**	**Wortanfang**	**vor Vokal**	**sonst**
TZ	TTZ, TZS, TSZ	4	4	4
T		3	3	3
UI	UJ, UY	0	1	–
U	UE	0	–	–
V		7	7	7
W		7	7	7
X		5	54	54
Y		1	–	–
ZDZ	ZDZH, ZHDZH	2	4	4
ZD	ZHD	2	43	43
ZH	ZS, ZSCH, ZSH	4	4	4
Z		4	4	4

Abkürzungsverzeichnis

CI
: Corporate Sector Information Systems and Services

CI/AF
: CI / Application Foundation & Security

CI/AFJ
: CI/AF / Java Development

EJB
: Enterprise JavaBeans

JDBC
: Java Database Connectivity

JPA
: Java Persistence API

JSF
: Java Server Faces

RDBMS
: Relationales Datenbankmanagementsystem

WAM
: Web Access Manager

Glossar

DVORAK-Tastatur
: Ein alternatives Tastatur-Layout, bei dem die Tasten nicht wie auf einer gewöhnlichen Tastatur angeordnet sind, sondern nach ergonomischen Gesichtspunkten.

Hashing-Algorithmus
: Ein Hashing-Algorithmus bildet eine große Menge von Daten auf eine kleine Menge ab. Dadurch entsteht eine Art Schlüssel, der wiederum als implizite Gruppierung dienen kann. Dies findet u. A. Anwendung bei Prüfsummenberechnungen oder digitalen Signaturen, aber eben auch bei der phonetischen Suche.

Lemma
: Die Linguistik bezeichnet die Grundform eines Wortes als Lemma (von griechisch λῆμμα (*lēmma*), „Annahme“). Es ist die Form des Wortes, die als Schlagwort in ein Wörterbuch aufgenommen wird.

Namensvariation (*engl.* name variation)
: Verschiedene Schreibweisen eines Namens oder gar verschiedene Namen können sich auf ein und dieselbe Person beziehen.

Namensüberladung (*engl.* name overloading)
: Mehrere verschiedene Personen können den gleichen Namen besitzen, d. h. ein Name bezieht sich nicht notwendigerweise auf genau eine Person.

Phonetik
: Die Phonetik beschäftigt sich mit der Aussprache und dem Klang von Wörtern. Der Begriff stammt vom griechischen φωνητικός (*phōnētikós*) ab, was *„zum Sprechen gehörig“* bedeutet und von φωνή (*phōnḗ*) *„Stimme, Laut, Klang, Ton“* abgeleitet ist.

QUERTY-Tastatur
: Tastatur-Layout, wie es in vielen englischsprachigen Ländern verwendet wird. Es ist benannt nach der Anordnung der ersten Tasten auf der oberen Buchstabenzeile von links: Q W E R T Y.

QUERTZ-Tastatur

Tastatur-Layout, wie es unter anderem in deutschsprachigen Ländern verwendet wird und Y und Z im Vergleich zur QUERTY-Tastatur vertauscht sind. Es ist benannt nach der Anordnung der ersten Tasten auf der oberen Buchstabenzeile von links: Q W E R T Z.

Scharfe Suche

Die *„scharfe“* Suche als Abgrenzung zur *„unscharfen“* Suche verlangt, dass Treffer vollständig, ðZeichen für Zeichen, dem Suchterm entsprechen.

Unscharfe Suche

Bei einer *„unscharfen“* Suche kann ein Treffer im Gegensatz zur *„scharfen“* Suche dem gesuchten Begriff auch nur ähnlich sein. Für die Berechnung der Ähnlichkeit gibt es mehrere Verfahren. Die für diese Arbeit relevanten wurden in Kapitel 2 vorgestellt.

Index

Literaturverzeichnis

[All08] D. Allen. *Seam in Action (In Action series)*. Manning Publications Co., Greenwich, CT, USA, 2008

[Arm00] M. Armstrong. An Overview of the Issues Related to the use of Personal Identifiers, 2000. URL http://www.statcan.gc.ca/pub/85-602-x/4193729-eng.pdf

[BDT09] A. Bronselaer, G. De Tré. A possibilistic approach to string comparison. *Trans. Fuz Sys.*, 17(1):208–223, 2009. doi:http://dx.doi.org/10.1109/TFUZZ.2008.2008025

[byt] Metaphone - a better Soundex. URL http://www.bytelife.nl/metaphone.htm

[Dam64] F. J. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, 1964. doi:http://doi.acm.org/10.1145/363958.363994

[DC94] M. W. Du, S. C. Chang. An Approach to Designing Very Fast Approximate String Matching Algorithms. *IEEE Trans. on Knowl. and Data Eng.*, 6(4):620–633, 1994. doi:http://dx.doi.org/10.1109/69.298177

[GJ02] C. Gibas, P. Jambeck. *Einführung in die Praktische Bioinformatik*. O'Reilly, Köln, 2002

[Ham50] R. W. Hamming. Error Detecting and Error Correcting Codes. Technical Report Vol. XXVI, No 2, S. 147 ff., Bell Systems, 1950

[HD80] P. A. V. Hall, G. R. Dowling. Approximate String Matching. *ACM Comput. Surv.*, 12(4):381–402, 1980. doi:http://doi.acm.org/10.1145/356827.356830

[HG05] E. Hatcher, O. Gospodnetic. *Lucene in Action (In Action series)*. Manning Publications Co., Greenwich, CT, USA, 2005

[hiba] Hibernate. URL http://www.hibernate.org/

[hibb] Hibernate Search. URL http://www.hibernate.org/subprojects/search.html

[HM05] R. Hardman, M. McLaughlin. *Expert Oracle PL/SQL*. Osborne ORACLE Press Series, 2005. URL http://www.oracle.com/technology/books/pdfs/expert%20oracle%20pl_sql%20ch%201.pdf

[KH07] M. Kehle, R. Hien. *Hibernate und die Java Persistence API*. entwickler.press, Frankfurt (Main), Deutschland, 2007

[Lev65] V. L. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, (1):8–17, 1965

[LR93] A. J. Lait, B. Randell. An Assessment of Name Matching Algorithms. 1993. URL http://homepages.cs.ncl.ac.uk/brian.randell/Genealogy/NameMatching.pdf

[luc] Apache Lucene. URL http://lucene.apache.org/

[MF82] M. Mor, A. S. Fraenkel. A hash code method for detecting and correcting spelling errors. *Commun. ACM*, 25(12):935–938, 1982. doi:http://doi.acm.org/10.1145/358728.358752

[Mok97] G. Mokotoff. Soundexing and Genealogy, 1997. URL http://www.avotaynu.com/soundex.html

[MRS08] C. D. Manning, P. Raghavan, H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, USA, 2008

[mys] MySQL 5.1 Referenzhandbuch. URL http://dev.mysql.com/doc/refman/5.1/de/index.html

[Nav01] G. Navarro. A Guided Tour to Approximate String Matching. *ACM Computing Surveys*, 33(1):31–88, 2001. URL http://www.egeen.ee/u/vilo/edu/2002-03/Tekstialgoritmid_I/Articles/Approximate/Navarro_Review_on_Approximate_Matching_p31-navarro.pdf

[omi] Omikron Firmenhomepage. URL http://www.omikron.net/

[ora] Oracle Database Documentation Library. URL http://www.oracle.com/technology/documentation/index.html

[Pos69] H.-J. Postel. Die Kölner Phonetik. Ein Verfahren zur Identifizierung von Personennamen auf der Grundlage der Gestaltanalyse. 1969. 19. Jahrgang, S. 925-931

[PT09] F. Patman, P. Thompson. Names: A New Frontier in Text Mining. In *Intelligence and Security Informatics*, volume 2665 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2009. doi:10.1007/3-540-44853-5_3. URL http://www.springerlink.com/content/tfpg9bhlq5d49vb2/

[Ros00] M. Rosenfelder. Hou tu pranownse Inglish, 2000. URL http://zompist.com/spell.html

[Sch04] S. Schüle. *Qualitätssicherung von Suchmaschinen*. Diplomarbeit, Fachhochschule Pforzheim, Hochschule für Gestaltung, Technik und Wirtschaft, 2004

[sol] Apache Solr. URL http://lucene.apache.org/solr/

[Tom03] S. Tomlinson. Lexical and Algorithmic Stemming Compared for 9 European Languages with Hummingbird SearchServer™at CLEF 2003. *CLEF*, 2003. URL http://www.clef-campaign.org/2003/WN_web/19.pdf

[Wik] Wikipedia: Schreibmaschinendistanz, 14. April 2009. URL http://de.wikipedia.org/w/index.php?title=Schreibmaschinendistanz&oldid=59000374

[Wil05] M. Wilz. *Aspekte der Kodierung phonetischer Ähnlichkeiten in deutschen Eigennamen*. Magisterarbeit, Universität zu Köln, Institut für Linguistik, 2005. URL http://www.uni-koeln.de/phil-fak/phonetik/Lehre/MA-Arbeiten/magister_wilz.pdf

[Win99] W. E. Winkler. The State of Record Linkage and Current Research Problems. Statistics of Income Division, Internal Revenue Service Publication R99/04, 1999. URL http://www.census.gov/srd/papers/pdf/rr99-04.pdf

Alle URLs wurden zuletzt am 03.05.2010 geprüft.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Dennis Scheck