FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Fachhochschule Aachen
Campus Jülich

Fachbereich: Medizintechnik und Technomathematik
Studiengang: Scientific Programming

Entwicklung und Optimierung einer Software zur performanten Speicherung und Verarbeitung von Massendaten für eine offene Systemlandschaft

Bachelorarbeit
von
Marcel Carlé

Aachen, 15. August 2012

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Bachelorarbeit mit dem Thema

„**Entwicklung und Optimierung einer Software zur performanten Speicherung und Verarbeitung von Massendaten für eine offene Systemlandschaft**“

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Unterschrift: _____

Aachen, den _____

Betreuung

Diese Arbeit wurde betreut von:

- **Erstprüfer:** Prof. Ulrich Stegelmann (Fachhochschule Aachen, Campus Jülich)
- **Zweitprüferin:** Dipl.-Math. (FH) Tamara Ostwald (sms eSolutions GmbH)

Die vorliegende Arbeit wurde in Zusammenarbeit mit der Firma sms eSolutions GmbH aus Düren angefertigt.

sms eSolutions GmbH
Willi-Bleicher-Straße 9
D-52353 Düren

Tel.: 02421 / 98 57 - 0
Fax.: 02421 / 98 57 - 999
EMail.: info@sms-esolutions.de

Geschäftsführung: Dipl.-Ing. Elmar Körner
Handelsregister-Nr.: 2749
UStID: DE 81298303

Zusammenfassung

In dieser Arbeit wird die Entwicklung einer Java EE Anwendung und die Optimierung einzelner Prozesse dieser Software beschrieben. Zunächst wird die Motivation für diese Software erläutert und anschließend die Anforderungen in einer Spezifikation zusammengeführt, aufgelistet und kurz beschrieben.
Eingesetzte Technologien, Schnittstellen und der Aufbau der Software, sowie der Datenbank, werden exemplarisch vorgestellt und teilweise mit Beispielen verdeutlicht.
Nachdem der Hintergrund ausreichend dargestellt wurde, wird auf einzelne beispielhafte Implementierungen und Optimierungen, die in der Software umgesetzt wurden, eingegangen.
Zum Schluss wird ein kurzes Fazit und ein Ausblick gegeben.

Inhaltsverzeichnis

1 Einleitung **1**

2 Spezifikation der Software **3**
2.1 Systemlandschaft . . . 3
2.2 Performance . . . 4
2.3 Routing-Aufträge . . . 5
2.4 Routing-Tabellen und Routing-Gruppen . . . 6
2.5 Routing-Änderungen . . . 7
2.6 Jobs . . . 7
2.7 Script-Engine . . . 7
2.8 Webschnittstelle . . . 7
2.9 Zusammenfassung . . . 8

3 Aufbau der Software **9**
3.1 Plattform und Technologie . . . 9
3.2 Application server . . . 9
3.3 Datenbank . . . 10
3.3.1 JPA - kurz vorgestellt . . . 10
3.3.2 Auswahl des Datenbanksystems . . . 12
3.3.3 Entity-Relationship Diagramm . . . 12
3.4 Webschnittstelle . . . 15
3.5 SOAP Schnittstelle . . . 17
3.6 Update Schnittstelle . . . 18

4 Implementierung und Optimierung **19**
4.1 Datenbankstruktur . . . 19
4.1.1 Aufbau der Daten . . . 19
4.1.2 Anzahl der Daten . . . 19
4.1.3 Anlegen von Indizes . . . 20
4.1.4 Fazit . . . 23
4.2 Caching . . . 23
4.2.1 Implementierung . . . 23
4.2.2 Erklärung des Quellcodes . . . 25
4.2.3 Auswertung . . . 25
4.3 SOAP-Schnittstelle . . . 26
4.3.1 Umsetzung . . . 26
4.3.2 Auszüge der Implementierungen . . . 27

4.4 Update Schnittstelle 29
4.4.1 Aufbau der übertragenen Daten 29
4.4.2 Performanceauswertung 30
4.5 Anwendung eines Routing-Auftrags 30
4.5.1 Implementierung 30
4.5.2 Optimierung 32
4.6 Konsolidierung 33
4.7 Test-Framework 35

5 Zusammenfassung **37**
5.1 Fazit 37
5.2 Ausblick 37

Anhang **38**
A.1 Das ER-Diagramm von Athena 38
Glossar 40
Abkürzungsverzeichnis 40

Abbildungsverzeichnis

1.1 Routing über einen Transit-Anbieter 1

2.1 Befüllung der Software (Athena) 3
2.2 Beispielablauf des täglichen Abgleichs mit einem TNB 4
2.3 Rücknahme eines alten Routing-Auftrags (Routingziel D002) 5
2.4 Beispielaufbau einer Routing-Gruppe 6

3.2 Beispiel einer einfachen Account-Tabelle 11
3.4 Beschreibung der Felder einer Routing-Tabelle 12
3.5 Beschreibung der Felder eines Routing-Eintrags 14
3.6 Beschreibung der Felder eines Routing-Auftrags 14
3.7 Beschreibung der Felder einer Routing-Änderung 15

4.1 Zusammenschluss von drei Routing-Einträgen 20
4.2 Beispiel eines B^+-Baumes für das Attribut *Fragment* 21
4.3 Darstellung des Index als einfachen Baum 22
4.4 Nassi-Shneiderman-Diagramm zum Anwenden eines Routing-Auftrags . . 31
4.5 Beispielhafte Darstellung des Verarbeitungsprozesses eines Routing-Auftrags 32
4.6 Beispielhafte Darstellung des Konsolidierungsprozesses nach Priorität . . 34
4.7 Beispielhafte Darstellung des Konsolidierungsprozesses nach Datum . . . 34

A.1 ER-Diagramm von Athena 39

Quellcodeverzeichnis

3.1 Beispiel einer einfachen Entität 10
3.3 Beispiel eines einfachen Managers 11
3.4 Beispiel einer *Hello World* Seite mittels JSF 16
3.5 Beispiel einer Managed Bean 17
3.8 Beispiel einer einfachen SOAP Anfrage 18

4.1 Meist benutztes SQL Statement zum Finden von Routing-Einträgen . . . 21
4.2 Index 21
4.3 Ausschnitt vom RoutingTableCache 24
4.4 Beispielaufruf einer Methode vom RoutingTableCache 25
4.5 Ausschnitt der XSD 27
4.6 Ausschnitt der WSDL 28
4.7 Ausschnitt der Klasse RoutingOrderServiceImpl 29
4.8 Aufteilung in mehrere Transaktionen und Leeren des Caches 33
4.9 Ausschnitt eines JUnit-Tests 35

1 Einleitung

Durch die Liberalisierung des deutschen Telekommunikationsmarktes im Jahre 1998 verlor die Deutsche Telekom AG ihre Monopolstellung für Sprachtelefoniedienste. Neue Anbieter von Telefoniediensten, so genannte Teilnehmernetzbetreiber (TNB), stiegen in den deutschen Markt ein. Die Bundesnetzagentur (BNetzA) legte ein Verfahren zum Austausch von Portierungsdaten fest. An dieses Verfahren sind alle TNB gebunden und tauschen seit dem ihre Portierungsdaten untereinander aus.
Unter Portierungsdaten versteht man eine Ansammlung von Rufnummer-Portierungen. Eine Rufnummer-Portierung entsteht, wenn ein Kunde seinen Anbieter wechselt und seine Rufnummer behalten möchte.

Bei dem Verfahren gibt es keine gemeinsame zentrale Datenbank für die Rufnummernzuordnung. Daher muss jeder TNB einen eigenen Datenbestand anhand der ausgetauschten Portierungsdaten aufbauen und pflegen. Diesen Datenbestand nutzt der TNB um die Anrufe seiner Kunden an den richtigen TNB weiterleiten zu können.
Kann er anhand seines Datenbestandes den Angerufenen einem TNB nicht zuordnen, so muss er den Anruf an einen Transit-Anbieter weiterleiten, welcher das Routing des Anrufs übernimmt.

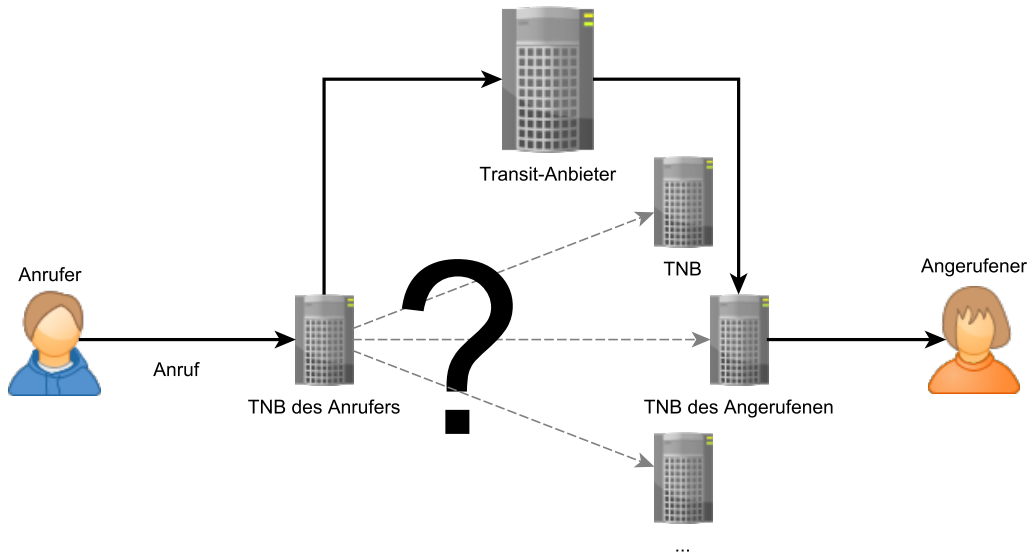

Abbildung 1.1: Routing über einen Transit-Anbieter

Leitet er den Anruf an den falschen TNB oder an einen Transit-Anbieter weiter, entstehen Interconnection-Gebühren von aktuell etwa einem halben Cent pro Minute[1].
Diese Gebühren erscheinen auf den ersten Blick vielleicht eher gering. Die TNB bieten ihren Kunden allerdings günstige Telefon- und Internet-Flatrates teilweise für insgesamt unter 20 Euro pro Monat an. Wird der Anruf nicht korrekt weitergeleitet, so bietet das folgende Szenario ein gutes Beispiel für die Kosten, welche dadurch entstehen können:

*Anrufer A telefoniert alle zwei Tage mit seiner Mutter und die Telefonate dauern im Schnitt jedes mal etwa 90 Minuten. Somit telefoniert er im Monat 15 * 90 Minuten = 1350 Minuten. Kennt sein Anbieter den Anbieter des Angerufenen nicht, kostet dies dem TNB 1350 Minuten * 0,5 Cent/Minute = 6,75€ im Monat.*

Das Beispiel verdeutlicht die Wichtigkeit eines guten Datenbestandes und zeigt auch die Einsparmöglichkeiten, welche sich dadurch ergeben können.

[1] „In der wichtigsten Tarifzone I (Verbindungsübergabe auf der untersten Netzebene) fallen demnach an Werktagen von 9 Uhr bis 18 Uhr netto 0,45 Cent pro Minute an“[teltarif]

2 Spezifikation der Software

Um diese Einsparmöglichkeiten für die TNB möglichst gut ausschöpfen zu können, entwickle ich für die sms eSolutions GmbH eine Software, welche einen möglichst optimalen Routing-Datenbestand besitzen soll. Dieser wird nur erreicht, wenn aus verschiedenen Systemen bzw. Datenquellen Routing-Informationen in das System einfließen.
Im Folgenden wird die Software Athena genannt. Der Name kommt von der griechischen Göttin Athene, welche als Göttin der Weisheit gilt.

2.1 Systemlandschaft

Um von verschiedenen Systemlandschaften die Routing-Informationen zu erhalten, bietet die Software einheitliche Schnittstellen. Über diese kann die Software von vielen Datenquellen befüllt und der Datenbestand optimiert werden.

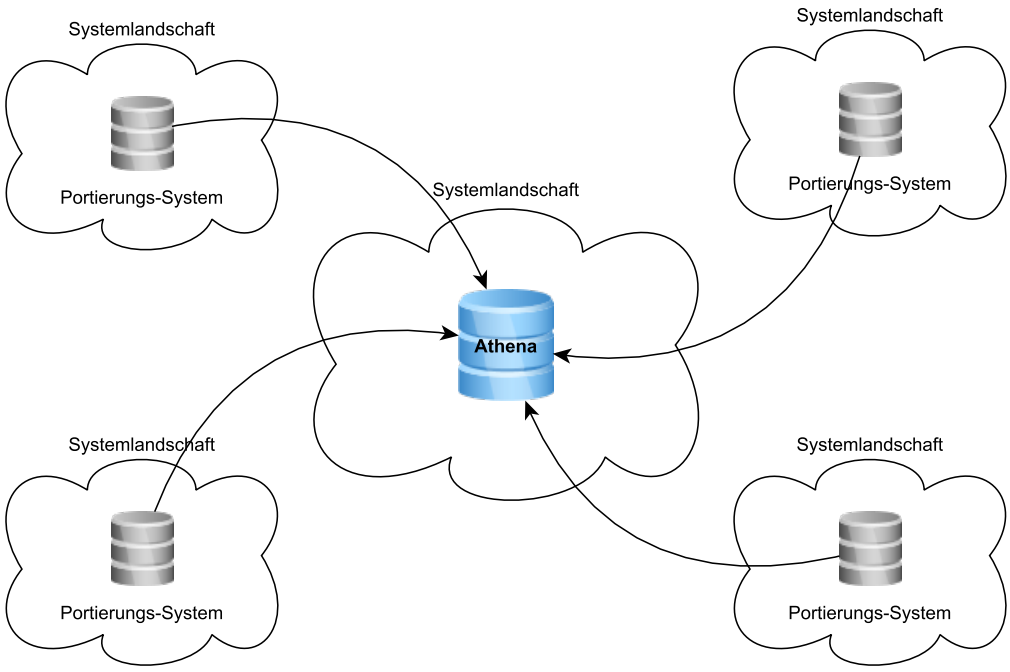

Abbildung 2.1: Befüllung der Software (Athena)

Mit Systemlandschaft sind hier unter anderem Betriebssysteme und Programmiersprachen gemeint. Die Berücksichtigung dieser Punkte ist notwendig, da die TNB auf verschiedenen (Betriebs-)Systemen arbeiten und die Portierungs-Softwareprodukte auch in verschiedenen Programmiersprachen geschrieben sind.

2.2 Performance

Neben einem guten Datenbestand und den Schnittstellen ist auch die Performance der Software ein wichtiger Aspekt.
Eine Vielzahl von Systemen muss die Software täglich mit mehreren tausend bis mehreren zehntausend Routing-Informationen befüllen können. Anschließend muss die Software anhand der neuen und der bereits vorhandenen Informationen den Datenbestand aktualisieren, konsolidieren und die so erhaltenen neuen Routing-Informationen exportieren.
Diese Prozesse sollten möglichst schnell abgearbeitet werden, da die TNB ansonsten mit veralteten Informationen die Anrufe weiterleiten müssen.

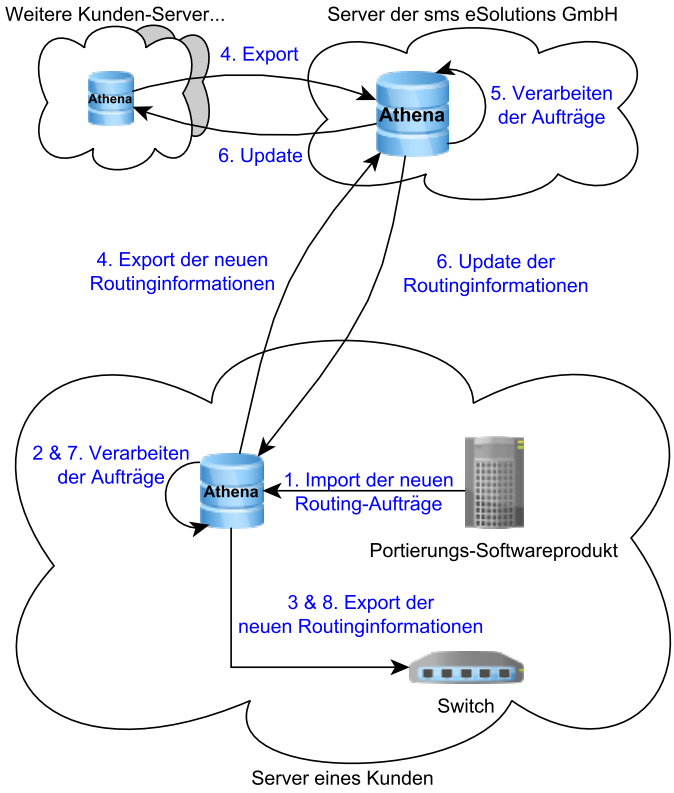

Abbildung 2.2: Beispielablauf des täglichen Abgleichs mit einem TNB

Um den Prozess weiter zu optimieren, kann Athena neben der Installation auf dem Server der sms eSolutions GmbH auch bei den teilnehmenden TNB vor Ort installiert werden. Dies ermöglicht den TNB ein tägliches Befüllen ihrer lokalen Installation mit neuen Routingdaten und damit ein zeitnahes Aktualisieren des Routings für die Anrufe. Gleichzeitig sendet die lokale Installation die neuen Informationen an die Software auf dem Server der sms eSolutions GmbH.
Sobald die Verarbeitung auf der serverseitigen Installation abgeschlossen ist, kann sich die lokale Installation den aktuellen Datenbestand herunterladen und das Routing erneut anpassen (siehe Abbildung 2.2).

2.3 Routing-Aufträge

Neue Routing-Informationen kommen in der Software als Routing-Auftrag an und werden im Zuge eines Verarbeitungsprozesses in Routing-Einträge überführt. Die Routing-Einträge spiegeln dabei immer nur den aktuellen Routing-Bestand wieder, wobei die Routing-Aufträge alle jemals erhaltenen Routing-Informationen enthalten.
Durch die Trennung zwischen Routing-Aufträgen und Routing-Einträgen bleibt die Datenmenge der Einträge im Vergleich zu den Aufträgen gering, was das spätere Auslesen erheblich beschleunigt.
Weiterhin können so beliebig alte Routing-Aufträge zurück genommen und anhand der übrigen Routing-Aufträge neu konsolidiert werden.
Die Grafik 2.3 zeigt, wie die Software die Routing-Einträge aktualisiert, wenn ein Routing-Auftrag zurück genommen wird.

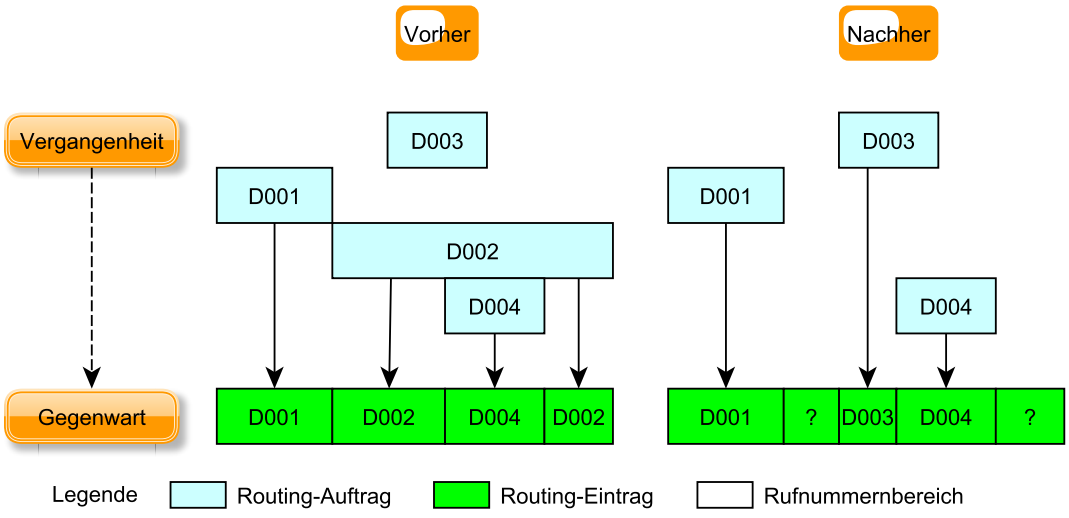

Abbildung 2.3: Rücknahme eines alten Routing-Auftrags (Routingziel D002)

2.4 Routing-Tabellen und Routing-Gruppen

Routing-Informationen werden in verschiedene Ebenen - im folgenden Routing-Tabellen genannt - gespeichert. Diese Aufteilung bewirkt eine einfache Unterscheidung der erhaltenen Routing-Informationen. Die Software kann so konfiguriert werden, dass beispielsweise alle noch nicht bestätigten (offenen) Portierungen in einer Routing-Tabelle und alle abgeschlossenen Portierungen in einer anderen gespeichert werden.

Welche Routing-Informationen aus welchen Routing-Tabellen für das Routing exportiert und genutzt werden, wird anhand von Routing-Gruppen bestimmt. Routing-Gruppen sind aufgebaut wie ein (Datei-)Baum (siehe Abbildung 2.4): Es gibt Routing-Gruppen (Ordner) und es gibt Routing-Tabellen (Dateien).
Für jede Gruppe wird ein Verarbeitungstyp festgelegt. Dieser gibt für den Konsolidierungsprozess an, ob die Unterelemente der Gruppe nach Priorität oder nach Datum ausgewertet werden.

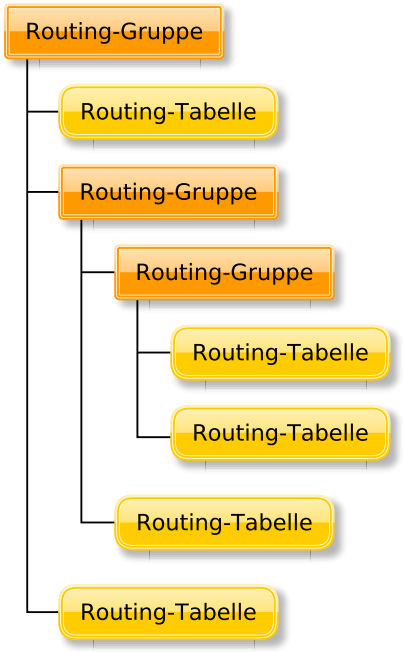

Abbildung 2.4: Beispielaufbau einer Routing-Gruppe

Priorität
Die Priorität spiegelt sich in der Position der Elemente im Baum wieder. Der Konsolidierungsprozess bestimmt das Routing-Ziel von oben (hohe Priorität) nach unten (niedrige Priorität). Haben zwei Routing-Tabellen unterschiedliche Informationen zu einer Rufnummer, so wird die Information aus der Tabelle mit der höheren Priorität ausgewertet.

Datum
Die Reihenfolge der Elemente im Baum ist nicht relevant. Der Konsolidierungsprozess arbeitet zwar auch von oben nach unten, überschreibt aber schon berechnete Routing-Informationen, wenn neuere gefunden wurden.

Sobald der Prozess auf eine Gruppe stößt, wird diese zuerst komplett ausgewertet und dann im weiteren Verlauf wie eine Routing-Tabelle verwendet.

2.5 Routing-Änderungen

Täglich müssen die neuen Routing-Informationen von Athena exportiert werden, damit die TNB ihr Routing anpassen können. Einige Switches machen jeden Tag einen Komplettabgleich der Routing-Daten, andere arbeiten mit einem täglichen Abgleich der veränderten Daten.
Mit den Routing-Einträgen kann jedoch nicht festgestellt werden, welche Informationen sich verändert haben. Daher ist es nötig, bei jeder Veränderung der Routing-Einträge eine Routing-Änderung zu erstellen. Anhand einer Routing-Änderung kann der Switch dann ein Routing entweder anpassen, löschen oder ein neues hinzufügen. Mit den Routing-Einträgen wäre nur letzteres möglich.

Ein weiterer Vorteil ist durch die Auslastungs- und Zeiteinsparung gegeben, da kein Komplettexport, sondern nur ein Teilexport der veränderten Daten stattfindet. Es entsteht weniger Auslastung für die Software und die Änderungen können schneller in einen Switch bei den TNBs geladen werden.

2.6 Jobs

Mit Hilfe von Jobs werden bestimmte Aufgaben in der Software automatisiert. Es können Routing-Aufträge angewendet, Updates von anderen Instanzen gefahren, Skripte ausgeführt und Routing-Gruppen konsolidiert werden.
Zu jedem Job gibt es einen oder mehrere Job-Trigger. Diese werden zu bestimmten Uhrzeiten oder nach der Fertigstellung bestimmter Jobs ausgelöst (engl. *triggered*).
Sobald ein Trigger ausgelöst wird, startet er seinen zugehörigen Job.

2.7 Script-Engine

Jeder TNB hat eigene Anforderungen an den Aufbau einer Exportdatei. Damit nicht für jeden neuen TNB die Software angepasst werden muss, verfügt sie über eine eigene Script-Engine, welche Skripte in der Sprache *JavaScript* ausführen kann.
Über diese Engine können jedem Kunden ein oder mehrere individuelle Exportskripte zur Verfügung gestellt werden.
Weitere automatisierte Aufgaben, wie zum Beispiel regelmäßig statistische Auswertungen, sind damit ebenfalls realisierbar.

2.8 Webschnittstelle

Um sämtliche Funktionen konfigurieren zu können, weist die Software eine Webschnittstelle auf. Sie bietet Möglichkeiten um Routing-Tabellen, Routing-Gruppen, Jobs und Skripte zu erstellen, zu konfigurieren und zu bearbeiten.
Außerdem können mit Hilfe der Webschnittstelle manuell Routing-Aufträge eingestellt

und angewendet, sowie Routing-Informationen abgefragt werden. Ein manuelles Starten und Stoppen der Jobs ist über die Webschnittstelle ebenfalls möglich.

2.9 Zusammenfassung

Fasst man die Anforderungen zusammen, entsteht eine Software, welche über mehrere Schnittstellen - u.a. eine Webschnittstelle - verfügt, die erhaltenen Daten komprimiert und optimiert speichert, sowie Routing-Informationen performant verarbeiten kann. Mit Jobs können Aufgaben automatisiert werden und mit der Script-Engine individuelle Exporte oder Auswertungen erzeugt werden.

3 Aufbau der Software

3.1 Plattform und Technologie

Als Grundlage der Software dient die Programmiersprache Java mit der Java EE (JEE) Plattform. Mittels JEE werden heutzutage viele Web-Anwendungen entwickelt und sie gilt als Konkurrent der .NET-Plattform von Microsoft. JEE ist dabei im Endeffekt zunächst nur die Spezifikation für eine Softwarearchitektur. Es basiert auf Standards und umfasst 33 verschiedene Java Specification Requests (JSRs) für die unterschiedlichsten Bereiche in der Anwendungs- und speziell der Webentwicklung mit Java[1].

Zu den JSRs gehören unter anderem Technologien wie Java Transaction API (JTA), Java Message Service (JMS), Java Persistence API (JPA), Enterprise JavaBeans (EJB), Context and Dependency Injection (CDI), Java API for XML Web Services (JAX-WS), JavaServer Faces (JSF) und viele weitere.
Durch die offenen Standards von JEE gibt es eine Vielzahl von kommerziellen und auch Open-Source Frameworks (zum Beispiel GlassFish, JBoss, Hibernate, Open JPA, Jersey, etc.), welche eben jene Standards implementieren und so zum Beispiel das *Transaction Handling* übernehmen[2].
Durch die Nutzung von JEE und Implementierung in Java ist die Software weitestgehend unabhängig von den eingesetzten Betriebssystemen. Die einzige Voraussetzung ist ein Betriebssystem, für welches die Java-Technologie vorhanden ist. Zu diesen zählen aktuell alle modernen Betriebssysteme, wie Windows, Mac OS, Solaris und Linux.

3.2 Application server

Zur Ausführung einer mit JEE geschriebenen Software, wird ein Anwendungsserver (engl. *Application server*) benötigt, welcher eine Laufzeitumgebung für die Anwendung bereitstellt. Über den Anwendungsserver werden dann die Webanwendungen bereitgestellt (engl. *deployed*). Zur Auswahl im Open-Source Segment stehen JBoss 7 und GlassFish 3.
Bei Athena fiel die Wahl auf GlassFish in der Version 3.1, da dieser die Referenzimplementierung (engl. *Reference Implementation*) (RI) für die JEE Technologie ist und schon für andere Anwendungen der sms eSolutions GmbH verwendet wird.

[1] „[...] Java EE is based on standards. It is an umbrella specification that bundles together a number of other JSR.“[Beginning Java EE 6, Seite 4, Absatz „Standards“]

[2] „Java EE provides open standards that are implemented by several commercial [...] or open source [...] frameworks for handling transactions, security, [...] and so on.“ [Beginning Java EE 6, Seite 4, Absatz „Standards“]

3.3 Datenbank

Um mit einer Datenbank aus Java heraus zu kommunizieren wird häufig die Java Database Connectivity (JDBC) Schnittstelle genutzt. Sie definiert „eine einheitliche Schnittstelle zu Datenbanken verschiedener Hersteller [...] und [ist] speziell auf relationale Datenbanken ausgerichtet“[Wikipedia, Java Database Connectivity]
Eine andere Möglichkeit ist mit der JPA und dem Hibernate Framework gegeben. Sie sind Schnittstellen um Plain Old Java Objects (POJOs) auf einzelne Tabellen in einer relationalen Datenbank abzubilden, was auch als Object-Relational Mapping (ORM) bezeichnet wird.

3.3.1 JPA - kurz vorgestellt

Mit dem Einsatz von JPA ist es in Java mittels der `@Entity` Annotation möglich mit einfachen POJOs Datenbanktabellen zu definieren. Dabei sind die Spalten (Attribute) durch die Variablen der Klasse definiert und eine Zeile (Tupel) in der Tabelle ist ein persistiertes, also gespeichertes, Objekt.
Das folgende Beispiel zeigt eine einfache Entität mit zwei Attributen:

```
@Entity
public class Account {

    private String name;
    private String password;

    @Basic
    public String getName() {
        return this.name;
    }

    public void setName(final String name) {
        this.name = name;
    }

    @Basic
    public String getPassword() {
        return this.password;
    }

    public void setPassword(final String password) {
        this.password = password;
    }
}
```

Quellcode 3.1: Beispiel einer einfachen Entität

In der Datenbank sieht die Tabelle dann wie folgt aus:

Account	
Name	**Password**
BspUser1	BspPassword1
BspUser2	BspPassword2
BspUser2	BspPassword2

Abbildung 3.2: Beispiel einer einfachen Account-Tabelle

Sobald die Anwendung in einem Anwendungsserver eingebunden wird, übernimmt dieser die Erstellung der Relationen, falls diese noch nicht existieren. Mittels einem `EntityManager` können Objekte in die Datenbank geschrieben, bzw. Abfragen und Updates durchgeführt werden. Solche Operationen erfolgen üblicherweise durch Enterprise JavaBeans, welche in Athena als Manager bezeichnet werden.

```
@Stateless
public class AccountManager {

    @PersistenceContext
    protected EntityManager entityManager;

    public Account findAccountById(final Long id) {
        try {
            return entityManager.find(Account.class, id);
        } catch (final NoResultException exception) {
            return null;
        }
    }
}
```

Quellcode 3.3: Beispiel eines einfachen Managers

Die Annotation `@Stateless` definiert den AccountManager als `Stateless Session-Bean`, welcher die Businesslogik implementiert. Session Beans kümmern sich um Operationen, wie zum Beispiel Berechnungen oder Zugriffe auf die Datenbank[3].
Mit `@PersistenceContext` wird der `EntityManager` eingebunden, über den die Datenbankoperationen ausgeführt werden.
Die Methode `findAccountById`, welche die Id eines `Account` Objektes übergeben bekommt, führt eine `Select`-Abfrage durch und liefert den passenden Account zurück.

[3] „A session bean class is any standard Java class that implements business logic.“[Beginning Java EE 6, Seite 217, Absatz „Bean Class“]

3.3.2 Auswahl des Datenbanksystems

Durch die Nutzung von JPA als Schnittstelle und Nutzung des Hibernate Frameworks im Hintergrund, stehen sehr viele relationale Datenbanken zur Auswahl. Auf Grund der hohen Performanceanforderungen von Athena, eignet sich die Oracle Datenbank am besten. Diese ist unter anderem für ihre analytischen Funktionen und Performance bekannt. Sie „skaliert sehr gut, unterstützt große Datenmengen und viele Plattformen“[Open-Source-Tag Magdeburg, Folie 12]. Daher setzt die sms eSolutions GmbH auf ihrem Server Oracle als Datenbank ein.
Die meisten Kunden werden ebenfalls Oracle als Datenbank verwenden, da sie bereits andere Software-Produkte der sms eSolutions GmbH unter Oracle nutzen. Für Bestandskunden erübrigt sich somit der Neukauf von Oracle und Schulung von Mitarbeitern.
Den Kunden bleibt es aber selbst überlassen, ob sie für ihre lokale Athena-Instanz eine andere relationale Datenbank verwenden möchten.

3.3.3 Entity-Relationship Diagramm

Das Entity-Relationship-Diagramm (kurz ER-Diagramm) im Anhang A.1 zeigt die vollständige Modellierung der Datenbankseite. Im Folgenden gehe ich auf einige der Entitäten ein und erläutere deren Aufbau.

Routing-Tabelle

RoutingTable	
Feld	**Beschreibung**
Id	Anhand dieser numerischen Id ist eine Routing-Tabelle eineindeutig zu identifizieren.
Name	Speichert einen eindeutigen Namen.
Description	Speichert eine Beschreibung der Routing-Tabelle. Diese hat rein informellen Charakter und dient den Anwendern der Software als Information, welche Routing-Informationen zu dieser Routing-Tabelle gehören.
RecordUpdates	Ein `Boolean` Feld, welches festlegt, ob für diese Routing-Tabelle Routing-Änderungen gespeichert werden sollen.
IgnoreSwitchDate	Ein `Boolean` Feld, welches festlegt, ob das `switchDate` an den zu dieser Routing-Tabelle zugehörigen Routing-Einträgen normalisiert, also auf ein festes Datum gesetzt werden soll. Dieses Flag bietet einen guten Geschwindigkeitsvorteil, jedoch gehen dadurch auch Informationen verloren.

Abbildung 3.4: Beschreibung der Felder einer Routing-Tabelle

Routing-Eintrag

RoutingEntry		
Feld	**Beschreibung**	**Beispiel**
Id	Anhand dieser numerischen Id ist ein Routing-Auftrag eineindeutig zu identifizieren.	1
RangeStart	Speichert eine 15-stellige Rufnummer. In Deutschland sind Rufnummern üblicherweise zwar nur bis zu 11 Stellen lang, aber international sind bis zu 15 Stellen möglich. Da es neben Einzel-Rufnummern auch Rufnummernblöcke gibt, wird zwischen Start- und Endnummer unterschieden. Handelt es sich um eine Einzel-Rufnummer, sind Start- und Endnummer identisch. In diesem Feld wird die Startnummer des Bereichs gespeichert.	246161000000000
RangeEnd	Speichert Äquivalent zu `RangeStart` eine 15-stellige Rufnummer, wobei hier die Endnummer gemeint ist.	246161099999999
Fragment	Speichert die ersten x Stellen der Rufnummer. Hier wird bewusst gegen die Normalform verstoßen und Daten, die in RangeStart und RangeEnd schon enthalten sind, nochmals gespeichert. Zur Zeit dieser Arbeit ist x noch auf vier festgelegt. Es ist aber möglich und geplant, je nach Größe des Datenbestandes, mehr als nur die ersten vier Stellen zu speichern. Dieses Feld bietet einen enormen Geschwindigkeitsvorteil.	2461
TargetType	Ein Feld um ein Java Enum zu speichern. Es gibt drei verschiedene Typen: `DEFAULT` bedeutet, dass beim Konsolidierungsprozess die nächste Routing-Tabelle berücksichtigt werden soll. `UNKNOWN` bedeutet, dass zu diesem Rufnummernbereich keine Informationen vorliegen. `DESTINATION` bedeutet, dass der Wert im Feld `TargetValue` das Routing-Ziel ist.	`DESTINATION`
TargetValue	Wird nur bei einem `TargetType` von `DESTINATION` gesetzt. Es gibt einen eindeutigen Code an, welcher das Routing-Ziel beschreibt.	D001

SwitchDate	Hier wird das Datum gespeichert, ab dem dieser Routing-Auftrag gültig ist.	30.07.2012
TableId	Speichert in Form einer Id den Verweis auf eine Routing-Tabelle. Über dieses Feld wird unterschieden, welcher Routing-Auftrag zu welcher Routing-Tabelle gehört.	1

Abbildung 3.5: Beschreibung der Felder eines Routing-Eintrags

Routing-Auftrag

Ein Routing-Auftrag ist sehr ähnlich aufgebaut wie ein Routing-Eintrag, speichert jedoch nicht das Datum, an dem er erstellt wurde.
Des Weiteren besitzt ein Routing-Auftrag folgende Felder:

RoutingOrder		
Feld	**Beschreibung**	**Beispiel**
...	...	...
Status	Kann drei verschiedene Werte annehmen: `PROCESS` besagt, dass der Auftrag noch verarbeitet werden muss. `PROCESSED` besagt, dass der Auftrag bereits verarbeitet wurde. `REVERT` besagt, dass der Auftrag zurück genommen werden soll.	`PROCESSED`
ProcessDate	Sobald ein Routing-Auftrag verarbeitet wurde und in den Status `PROCESSED` wechselt, wird hier der Zeitstempel der Verarbeitung gespeichert.	31.07.2012
Remark	Für manuell erstellte Routing-Aufträge können die Anwender einen Kommentar zu dem Routing-Auftrag abgeben.	Falsches Routing behoben.
CreatedBy	Speichert den Athena-Nutzer, über welchen dieser Routing-Auftrag in das System kam.	ICCS

Abbildung 3.6: Beschreibung der Felder eines Routing-Auftrags

Routing-Änderung

Ähnlich einem Routing-Eintrag und einem Routing-Auftrag ist eine Routing-Änderung aufgebaut. An ihr wird nicht das Erstellungsdatum gespeichert, sondern eine Art Routing-Kommando:

RoutingUpdate		
Feld	**Beschreibung**	**Beispiel**
...	...	...
Command	Kann drei verschiedene Werte annehmen: `ADD` besagt, dass ein neuer Routing-Eintrag angelegt werden soll. `DELETE` besagt, dass der Routing-Eintrag gelöscht werden soll. `UPDATE` besagt, dass der Routing-Eintrag aktualisiert werden soll.	`ADD`

Abbildung 3.7: Beschreibung der Felder einer Routing-Änderung

3.4 Webschnittstelle

Zur Darstellung über die Webschnittstelle wird das JSF Framework verwendet. Dies entstand auf Grund der Limitierungen durch JavaServer Pages (JSP) und basiert auf Servlets und JSP. Es erlaubt den Entwicklern die Graphical User Interface (GUI) mit graphischen Komponenten zu bauen, ohne sich viele Gedanken um Anfrage, Antwort und Hypertext Markup Language (HTML) machen zu müssen[4].

Mit Hilfe von Managed Beans wird die Interaktion zwischen der Business Logik (Java) und der GUI, sowie die Navigation zwischen den Seiten realisiert.

Eingesetzt wird die Referenzimplementierung *Mojarra*, welche bei GlassFish mitgeliefert wird, sowie das Komponentenframework *RichFaces*, mit welchem die Funktionalität der JSF-Implementationen erweitert wird.

Das folgende Beispiel zeigt die Implementierung einer einfachen JSF Seite.

[4] „JSF allows developers to think in terms of components, events, managed beans, and their interactions, instead of requests, responses, and markup language.“[Beginning Java EE 6, Seite 277, Kapitel „JavaServer Faces“]

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
         "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:a4j="http://richfaces.org/a4j">
    <h:head>
        <title>Hello World</title>
    </h:head>
    <h:body>
        <h:form>
            <h:outputText value="Name: " />
            <h:inputText value="#{testController.name}" >
                <a4j:ajax event="keyup" render="helloText" />
            </h:inputText>
            <br/>
            <h:outputText id="helloText"
                value="Hello #{testController.name}!" />
        </h:form>
    </h:body>
</html>
```

Quellcode 3.4: Beispiel einer *Hello World* Seite mittels JSF

Die erste Zeile (`<?xml version='1.0' encoding='UTF-8' ?>`) gibt an, dass es sich um ein Extensible Markup Language (XML) konformes HTML handelt, also jeder Tag geschlossen werden muss.
Der sonstige Aufbau ist wie in HTML üblich: Zuerst die Dokumenttyp-Deklaration (`<!DOCTYPE >`), dann das HTML-Wurzelelement (`<html>`). Darunter der Header (`<h:head>`) für die so genannten Kopfdaten (zum Beispiel der Titel) und zum Schluss der Body (`<h:body>`) für den Inhalt.

Im Body wird hier ein einfaches Formular erstellt, welches drei graphische Komponenten beinhaltet:

- Einen konstanten Text: "Name: "
- Ein Eingabefeld für Text, welches in das Attribut *name* aus der Managed Bean *TestController* schreibt.
- Ein variabler Text, welcher auf das Attribut *name* aus der Managed Bean *TestController* zugreift und es ausgibt.

Dazu gibt es eine nichtgraphische Komponente aus *RichFaces*, welche Ajax-Anfragen beim `keyup`-Event des Eingabefeldes ausführt. Dabei wird der eingegebene Text in das Attribut *name* gespeichert und die Komponente mit der Id *helloText* neu gezeichnet (engl. *rendered*).

Im Folgenden ist die benötigte Managed Bean für das *Hello World*-Beispiel dargestellt.

```
@ViewScoped
@ManagedBean
public class TestController {

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Quellcode 3.5: Beispiel einer Managed Bean

Die Annotation `@ViewScoped` deklariert den Scope dieser Bean. Solange der Benutzer auf der aktuellen Seite bleibt, sind die Daten in der Bean persistiert, wechselt er die Seite, wird die Bean gelöscht (engl. *cleared*).

3.5 SOAP Schnittstelle

Simple Object Access Protocol (SOAP) ist ein einfaches, XML-basiertes Protokoll um Informationen über Hypertext Transfer Protocol (HTTP) auszutauschen[5].
Der SOAP-Server definiert die Schnittstelle und legt das Format der Anfrage- und Antwortnachrichten fest. Anhand dieser kann die Clientanwendung die Business Methoden mit den korrekten Parametern aufrufen und bekommt eine entsprechende Antwort, auf die sie dann reagieren kann.
Dabei sind die Programmiersprachen, in der Server und Client geschrieben sind, voneinander unabhängig und können somit in unterschiedlichen Sprachen entwickelt werden.
SOAP ist von verschiedenen Technologien abhängig. Vor allem von den folgenden drei:

- XML ist die Basis, auf der viele Webservices beruhen und welche durch eine XML Schema Definition (XSD) definiert ist.
- Web Services Description Language (WSDL) definiert die Schnittstelle und die Business Methoden mit ihren Parametern und Rückgabewerten.
- HTTP, welches das meist verbreitete Protokoll zur Übertragung von Daten im Internet ist.

Hier ist ein Beispiel einer SOAP Anfrage:

[5] „SOAP is a simple XML-based protocol to let applications exchange information over HTTP.“[w3schools, SOAP]

```
<soapenv:Envelope
 xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:ser="http://service.athena.sms.de/"
 xmlns:ser1="http://service.xml.athena.sms.de/">
   <soapenv:Header/>
   <soapenv:Body>
      <ser:addRoutingOrdersRequest>
         <auth>
            <ser1:name>UserName</ser1:name>
            <ser1:password>Password</ser1:password>
         </auth>
         <routingOrders>
            <routingOrder>
               <ser1:rangeStart>246161000000000</ser1:rangeStart>
               <ser1:rangeEnd>246161099999999</ser1:rangeEnd>
               <ser1:targetType>DESTINATION</ser1:targetType>
               <ser1:targetValue>D001</ser1:targetValue>
               <ser1:table>PDA-Portierungen</ser1:table>
               <ser1:switchDate>
                2012-07-18T00:00:00.000+02:00
               </ser1:switchDate>
            </routingOrder>
         </routingOrders>
      </ser:addRoutingOrdersRequest>
   </soapenv:Body>
</soapenv:Envelope>
```

Quellcode 3.8: Beispiel einer einfachen SOAP Anfrage

Über die SOAP Schnittstelle wird Athena täglich mit Routing-Informationen aus den verschiedensten Systemen versorgt. Dabei müssen nur die täglich neuen Routing-Informationen übertragen werden, wodurch eine - im Vergleich zum initialen Import - kleinere Datenmenge übertragen werden muss.

3.6 Update Schnittstelle

Bei einer Übertragung über SOAP werden durch den XML-Aufbau deutlich mehr Daten übertragen, als eigentlich benötigt werden (siehe Quellcode 3.8). Aus diesem Grund eignet sich eine SOAP Schnittstelle nicht für eine Übertragung von Massendaten. Weil die Update Schnittstelle jedoch nur für die Kommunikation zwischen den Athena Systemen genutzt wird und keine andere Software diese Schnittstelle implementiert, wurde ein eigenes Protokoll geschrieben, welches die Übertragungsgröße im Vergleich zur SOAP Schnittstelle deutlich reduziert.
Wie dieses Protokoll aussieht und funktioniert wird im weiteren Verlauf dieser Arbeit erläutert.

4 Implementierung und Optimierung

Im folgenden Teil meiner Bachelorarbeit werde ich auf die verschiedenen Möglichkeiten der Optimierungen eingehen und diese erläutern. Ich werde Implementationen an Beispielen zeigen.

4.1 Datenbankstruktur

Um eine performante Verarbeitung von Massendaten gewährleisten zu können, müssen die Daten schnell in eine Datenbank gespeichert und zum Zeitpunkt der Verarbeitung sehr schnell aus dieser wieder ausgelesen werden. Dazu kann man die Datenbankzugriffe mit Indizes beschleunigen, aber auch schon der Aufbau der gespeicherten Daten spielt eine wichtige Rolle.

4.1.1 Aufbau der Daten

Zunächst einmal muss überlegt werden, welche Daten die Software benötigt. Nimmt man als Beispiel das Portierungs-System Inter Carrier Communication System (ICCS) von der sms eSolutions GmbH, so besitzt eine Portierung in der Datenbank viele Felder, die Athena nicht benötigt. Insgesamt besteht eine Portierung im ICCS aus über 27 Attributen. Dazu zählen zum Beispiel der TNB, von dem die Rufnummer ursprünglich stammt, oder auch aus welchen Portierungs-Meldungen sich die Portierung zusammensetzt.

Diese Informationen spielen im Athena keine Rolle und entfallen daher. Wie die Abbildung 3.5 auf Seite 14 zeigt, wurden die Daten im Athena so weit wie möglich komprimiert und somit besteht ein Routing-Eintrag nur noch aus acht Attributen.

4.1.2 Anzahl der Daten

Die Menge der Daten in einer Datenbank wirkt sich auf die Laufzeiten von Structured Query Language (SQL) Statements aus. Aus diesem Grund werden Daten von Athena weitestgehend zusammengefasst. Grenzt ein Routing-Eintrag an einen anderen Routing-Eintrag, welcher dasselbe Routing-Ziel hat, so können diese beiden Einträge zu einem einzigen zusammengefasst werden. Dies kann mit beliebig vielen aneinander grenzenden Routing-Einträgen gemacht werden (siehe Abbildung 4.1).
Durch die Komprimierung der Routing-Einträge reduziert sich der Datenbestand um annähernd 25%. Bei sequentiellen Abfragen wird die benötigte Zeit daher ebenfalls um etwa 25% reduziert, bei logarithmischen um nur wenige Prozent.

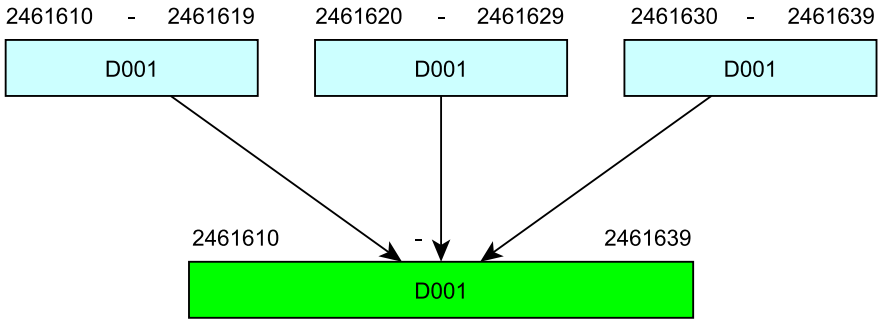

Abbildung 4.1: Zusammenschluss von drei Routing-Einträgen

4.1.3 Anlegen von Indizes

Auffällig ist das Attribut `Fragment` eines Routing-Eintrags. Dieses verstößt gegen die Normalformen, da die Daten schon in den Attributen `RangeStart` und `RangeEnd` enthalten sind und erhöht - wie der Name schon vermuten lässt - die Fragmentierung.
Wie in Abbildung 3.5 beschrieben, ist dieses Feld zur Zeit dieser Arbeit auf die ersten vier Stellen der Rufnummer festgelegt. Die Fragmentierung bleibt durch die geringe Anzahl Stellen ebenfalls sehr gering und wirkt sich kaum auf die Performance aus.

Durch die Nutzung von Datenbankindizes lässt sich mit Hilfe dieses Attributs jedoch eine erhebliche Performancesteigerung erzielen.

Ein Index wird in Datenbanken sehr häufig verwendet, um Abfragen und Sortiervorgänge nach bestimmten Feldern zu beschleunigen. In der Regel verwenden die Datenbankmanagementsysteme (DBMS) für Indizes B^+-Bäume, welche eine logarithmische Komplexität bieten (siehe dazu Abbildung 4.2). Ohne die Verwendung von Indizes müsste das DBMS die Spalte sequentiell durchsuchen[1].

Um die richtigen Indizes anlegen und dadurch die Geschwindigkeit optimieren zu können, müssen die zeitkritischen und häufig ausgeführten SQL Statements gefunden, analysiert und ausgewertet werden. Anschließend muss für jedes Statement ein eigener, geeigneter Index gefunden und erstellt werden.
Nicht selten ist man auf die *trial-and-error*-Methode angewiesen, da es von Datenbank zu Datenbank Unterschiede in der Handhabung von Indizes gibt. Nicht alle Datenbanken können Indizes gleichsam effizient nutzen.

[1] „In der Regel finden hier B^+-Bäume Anwendung. Ohne Index müsste die Spalte sequentiell durchsucht werden, während eine Suche mit Hilfe des Baums nur logarithmische Komplexität hat.“ [Wikipedia, Datenbankindex]

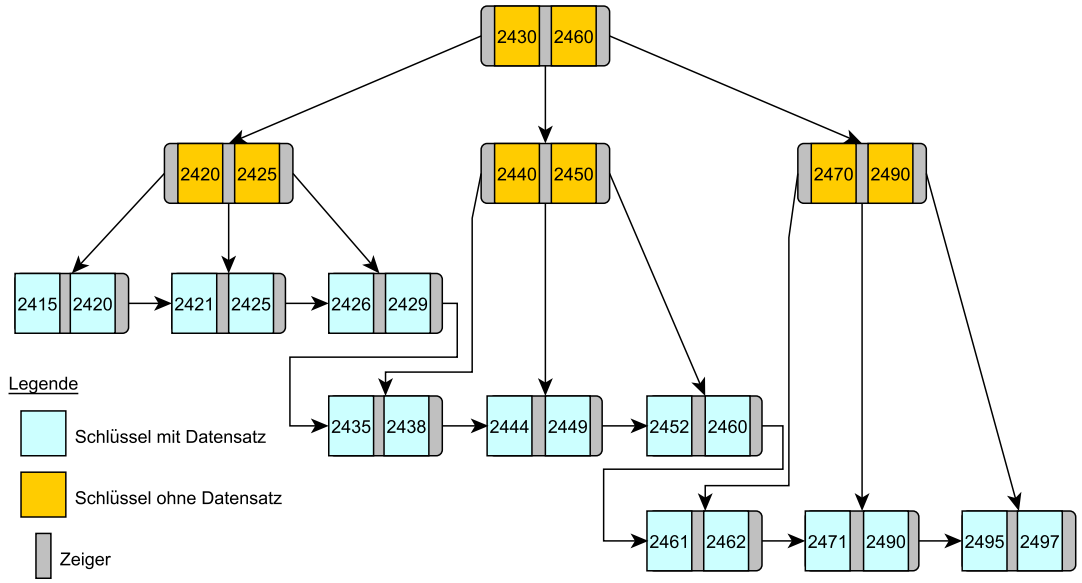

Abbildung 4.2: Beispiel eines B^+-Baumes für das Attribut *Fragment*

Statement

Das folgende Statement wird bei jeder Anwendung eines Routing-Auftrags verwendet. Es werden alle vorhandenen Routing-Einträge in einer Routing-Tabelle gesucht, welche den Rufnummernbereich des Routing-Auftrags überschneiden bzw. daran angrenzen. Dies wird benötigt, um die Daten so weit wie möglich zusammenfassen zu können.

```
select re.* from RoutingEntry re where re.table_id = ?
       AND re.rangeEnd >= ?
       AND re.rangeStart <= ?
       AND re.fragment = ?
ORDER BY re.rangeStart ASC
```

Quellcode 4.1: Meist benutztes SQL Statement zum Finden von Routing-Einträgen

Index

Bei diesem Statement wird nach den Attributen `table_id`, `rangeEnd`, `rangeStart` und `fragment` gefiltert. Erstellt man für dieses Statement keinen Index benötigt diese Abfrage mehrere Sekunden (bei einem Datenbestand von ca. 20 Millionen Routing-Einträgen). Mit dem folgenden Index sind es nur wenige Millisekunden.

```
create index IDX_RE_FRG_TID_BEG_END
on RoutingEntry (fragment, table_id, rangeStart, rangeEnd)
```

Quellcode 4.2: Index

Zur vereinfachten Darstellung, stellt die Abbildung 4.3 den Index als einfachen Baum dar.

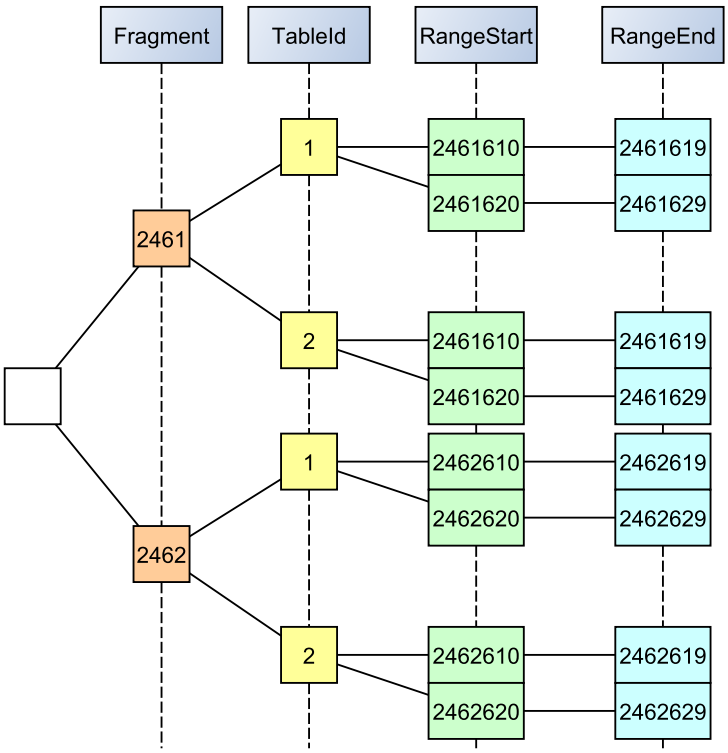

Abbildung 4.3: Darstellung des Index als einfachen Baum

Nachteile von Indizes

Man fragt sich natürlich sofort: Warum nicht für jedes SQL Statement einen Index anlegen? Welche Nachteile gibt es?
Die Antwort hängt immer davon ab, wie die Software eingesetzt wird, wie viel Aufwand in die Optimierung gesteckt werden soll und wie viel Nutzen am Ende erzeugt wird.
Der wichtigste Punkt ist aber, dass jeder Index den Einfüge-, Aktualisierungs-, und Löschprozess von Daten verlangsamt und sich der Speicherbedarf pro Index erhöht[2].

[2] „Erstens belegen Indexdateien Speicherplatz, und mehrere Indizes entsprechend mehr Speicherplatz. [...] Zweitens beschleunigen Indizes zwar die Suche, verlangsamen aber Einfüge- und Löschoperationen ebenso wie die Aktualisierung von Werten in indizierten Spalten (d.h. die meisten Operationen, bei denen etwas geschrieben wird), weil beim Schreiben jetzt nicht nur die Datenzeile berücksichtigt werden muss, sondern häufig auch der Index“[TecChannel, Indizes richtig einsetzen]

Außerdem kostet es Aufwand, jedes Statement zu analysieren und einen guten Index dafür zu finden.

4.1.4 Fazit

Mit Indizes lassen sich SQL Statements teilweise sehr deutlich beschleunigen, jedoch verringern sie gleichzeitig die Performance beim Schreiben.
Das Attribut `Fragment` eines Routing-Eintrags wird nur für Indizes benutzt, da es die Datenmenge gut in einzelne Gruppen unterteilt, welche nicht zu groß und nicht zu klein sind. Dabei relativiert die so erreichte Performancesteigerung die Mehrdaten die dadurch entstehen.
Man könnte nun aus den Attributen `RangeStart` und `RangeEnd` die ersten vier Ziffern entfernen, was aber bedeuten würde, dass nach jedem Selektieren der Daten die eigentliche Nummer zusammengesetzt werden müsste. Zugegeben, die benötigte Zeit dafür ist sehr gering, der Aufwand die Software darauf umzuschreiben, wäre jedoch deutlich höher.

Die Optimierungen, die hier beispielhaft an den Routing-Einträgen demonstriert wurden, sind unter anderem auch bei den Routing-Aufträgen umgesetzt worden.

4.2 Caching

Bei der Verarbeitung von Routing-Aufträgen, der Konsolidierung von Routing-Einträgen oder auch dem Import von Routing-Aufträgen wird häufig auf die gleichen Attribute der Routing-Tabelle zugegriffen.
Man nehme als Beispiel den Import von Routing-Aufträgen über die SOAP-Schnittstelle. Dabei wird, wie im Quellcode 3.8 zu sehen ist, der Name der Routing-Tabelle übertragen. In der Datenbank wird ein Routing-Auftrag jedoch nicht über den Namen mit einer Routing-Tabelle verknüpft, sondern über deren Id.
Es müsste also bei jedem erhaltenen Datensatz eine SQL-Abfrage ausgelöst werden, um die Id der Routing-Tabelle zu erhalten. Diese Abfragen würden jedes Mal Zeit in Anspruch nehmen und die Prozesse dadurch verlangsamen.

4.2.1 Implementierung

Mit Hilfe einer einfachen Klasse, welche sich die benötigten Daten und Zuordnungen zwischenspeichert, können in den meisten Fällen die Datenbankabfragen umgangen werden. Damit es von dieser Klasse nicht mehrere Instanzen gibt, kann man es in JEE als `Singleton` annotieren. Dadurch ist sicher gestellt, dass jede Bean dieselbe Instanz benutzt.

```
@Singleton
public class RoutingTableCache {
    @Inject
    private RoutingTableManager routingTableManager;
    private final Map<String, Long> tableIdsByName =
                                new HashMap<String, Long>();
    /**
     * Liefert die Id der Tabelle mit dem gegebenen Namen.
     * @param name Name der Tabelle
     * @return Id der Tabelle
     */
    public Long getTableIdByName(final String name) {
        final Long result = tableIdsByName.get(name);
        if (result == null) {
            final RoutingTable routingTable =
                cacheTable(routingTableManager.findRoutingTableByName(
                                                                name));
            return (routingTable == null) ? null : routingTable.getId();
        } else {
            return result;
        }
    }
    /**
     * Laedt eine Tabelle aus der Datenbank und cached die benoetigten
     * Daten.
     * @param routingTable RoutingTable-Werte
     * @return RoutingTable
     */
    private RoutingTable cacheTable(final RoutingTable routingTable) {
        if (routingTable != null) {
            tableIdsByName.put(routingTable.getName(),
                                    routingTable.getId());
        }
        return routingTable;
    }
    /**
     * Setzt den Cache zurueck.
     */
    public void reset() {
        tableIdsByName.clear();
    }
}
```

Quellcode 4.3: Ausschnitt vom RoutingTableCache

4.2.2 Erklärung des Quellcodes

Der Quellcode 4.3 zeigt einen kleinen Ausschnitt des `RoutingTableCache`. Dieser besteht hier im Wesentlichen aus drei Methoden:

- `cacheTable` Wird intern aufgerufen, wenn nach einer Routing-Tabelle gesucht wird, welche noch nicht *gecached* wurde. Wenn eine Routing-Tabelle gefunden wurde, wird der Map `tableIdsByName` ein neuer Eintrag hinzugefügt und die Routing-Tabelle zurück gegeben.
- `getTableIdByName`
  Bekommt einen Namen übergeben und sucht in der Map `tableIdsByName` nach einem passenden Eintrag. Wird einer gefunden, wird die erhaltene Id zurück gegeben. Wenn kein Eintrag gefunden wird, wird die Methode `cacheTable` aufgerufen. Liefert diese eine Routing-Tabelle zurück, wird die Id der Tabelle zurück gegeben, sonst `null`.
- `reset` Wird aufgerufen, wenn eine Routing-Tabelle bearbeitet, gelöscht oder neu angelegt wurde. Leert die Map `tableIdsByName`.

Beim Import von Routing-Aufträgen wird bei jedem Eintrag auf die Methode `getTableIdByName` vom `RoutingTableCache` zugegriffen.

Im vollständigen Quellcode werden zusätzlich zur Zuordnung `Name -> Id` die Zuordnungen `Id -> Name`, `Id -> RecordUpdates` und `Id -> IgnoreSwitchDate` in jeweils einer eigenen Map gespeichert und bieten jeweils eine eigene *getter*-Methode.

Aufgerufen wird die Methode `getTableIdByName` in den Beans zum Beispiel wie folgt:

```
@Stateless
public class SampleManager {
    @Inject
    private RoutingTableCache routingTableCache;

    public void doSomething(final String tableName) {
        ...
        final Long tableId = routingTableCache.getTableIdByName(tableName);
        ...
    }
}
```

Quellcode 4.4: Beispielaufruf einer Methode vom RoutingTableCache

4.2.3 Auswertung

Eine Performancesteigerung wird erst bei dem zweiten Aufruf der Methode erreicht, da beim ersten Aufruf weiterhin eine SQL Abfrage ausgeführt werden muss. Beim zweiten

Aufruf wird dann das Ergebnis aus der Map im Speicher zurück gegeben.
Dadurch wird eine Performancesteigerung von ein bis zwei Millisekunden pro Datensatz erreicht.
Dies bedeutet eine Einsparung von etwas mehr als 11 Stunden bei einem Import von 20 Millionen Datensätzen.

$$2 \frac{\text{ms}}{\text{Datensatz}} * 20.000.000 \text{ Datensätze} = 40.000 \text{ Sekunden} \approx 11 \text{ Stunden}$$

Üblicherweise wird eine solche Masse an Daten nur einmal in Athena importiert. Bei den täglichen Imports werden nur einige zehntausend Routing-Informationen übertragen.

Das Caching wird aber auch bei weiteren Prozessen eingesetzt, welche mit dem aktuell vorhandenen Datenbestand in der Datenbank arbeiten. Dieser umfasst zur Zeit dieser Arbeit etwa 20 Millionen Routing-Einträge seitens der sms eSolutions GmbH.

4.3 SOAP-Schnittstelle

Eine SOAP Schnittstelle ist eine standardisierte Webschnittstelle. Sie wird in Athena verwendet, um neue Routing-Aufträge in die Software zu importieren, alte Aufträge zurück zu nehmen und um Routing-Informationen abzufragen.

4.3.1 Umsetzung

Es gibt zwei verschiedenen Varianten um eine SOAP Schnittstelle zu erstellen. Beide Varianten besitzen einen ähnlichen Aufwand in der Implementierung und jeweils eigene Vor- und Nachteile.

- **Code-First**
  Hierbei wird der Java-Code manuell geschrieben und aus den Klassen und Methoden beim Kompilieren die WSDL generiert.

  Vorteile: Einfache POJOs, einfach zu schreiben
  Nachteile: Viel Java-Code, Einfügen von Kommentaren in die WSDL nicht möglich

- **Contract-First**
  Hierbei wird die WSDL manuell geschrieben und daraus beim Kompilieren die Java-Klassen mit den jeweiligen Methoden generiert.

  Vorteile: Wenig XML-Code, Einfügen von Kommentaren in die WSDL möglich
  Nachteile: Kenntnisse über WSDLs erforderlich

In Athena wird mit der *Contract-First* Variante gearbeitet. Wie oben beschrieben, wird dabei eine WSDL Datei manuell erstellt. In Athena werden die Datentypen, also die Parameter die den Methoden übergeben werden und von diesen zurück gegeben werden, in eine XSD ausgelagert.

4.3.2 Auszüge der Implementierungen

XML

Athena stellt vier Methoden über SOAP bereit:

- *addRoutingOrders*
  Dient zum Import von Routing-Aufträgen.
- *revertRoutingOrders*
  Dient zur Rücknahme von Routing-Aufträgen.
- *listRoutingInfosByTable*
  Dient zur Abfrage von Routing-Informationen aus einer einzelnen Routing-Tabelle.
- *listRoutingInfosByGroup*
  Dient zur Abfrage von Routing-Informationen aus einer Routing-Gruppe. Hierbei wird die Routing-Gruppe *on-the-fly* konsolidiert.

Der folgende Auszug aus der XSD zeigt, wie ein Datentyp definiert wird.

```
<xs:complexType name="authData">
  <xs:all>
    <xs:element name="name" type="xs:string" minOccurs="1" maxOccurs="1">
      <xs:annotation>
        <xs:documentation>
          Der Name des Accounts, in dessen Kontext die Aktion
          durchgefuehrt werden soll.
        </xs:documentation>
      </xs:annotation>
    </xs:element>
    <xs:element name="password" type="xs:string" minOccurs="1"
        maxOccurs="1">
      <xs:annotation>
        <xs:documentation>
          Das Passwort des Accounts.
        </xs:documentation>
      </xs:annotation>
    </xs:element>
  </xs:all>
</xs:complexType>
```

Quellcode 4.5: Ausschnitt der XSD

Der Quellcode 4.5 zeigt die Deklaration des Datentyps `authData`, welcher für die Anmeldung benötigt wird. Der Datentyp besitzt die Felder:

- `name` für den Benutzernamen
- `password` für das Passwort

Beide Felder sind vom Typ *String* (`type="xs:string"`) und müssen gesetzt sein (`minOccurs="1"` und `maxOccurs="1"`).

Im folgenden Auszug aus der WSDL werden die verfügbaren Methoden und deren Ein- und Ausgabeparameter festgelegt.

```
<wsdl:portType name="routingOrderPortType">
    <wsdl:operation name="addRoutingOrders">
        <wsdl:input message="tns:addRoutingOrdersInput" />
        <wsdl:output message="tns:addRoutingOrdersOutput" />
    </wsdl:operation>
    <wsdl:operation name="revertRoutingOrders">
        <wsdl:input message="tns:revertRoutingOrdersInput" />
        <wsdl:output message="tns:revertRoutingOrdersOutput" />
    </wsdl:operation>
    <wsdl:operation name="listRoutingInfosByTable">
        <wsdl:input message="tns:listRoutingInfosByTableInput" />
        <wsdl:output message="tns:listRoutingInfosByTableOutput" />
    </wsdl:operation>
    <wsdl:operation name="listRoutingInfosByGroup">
        <wsdl:input message="tns:listRoutingInfosByGroupInput" />
        <wsdl:output message="tns:listRoutingInfosByGroupOutput" />
    </wsdl:operation>
</wsdl:portType>
```

Quellcode 4.6: Ausschnitt der WSDL

Java

Zusätzlich zu der WSDL und der XSD muss die Business Logik noch in Java implementiert werden. In Athena wird dies durch die Klasse *RoutingOrderServiceImpl* übernommen. Diese erbt von einer aus der WSDL generierten Klasse und implementiert die vier deklarierten Methoden.

```
@WebService(serviceName = "routingOrderService",
        portName = "routingOrderPort",
        targetNamespace = "http://service.athena.sms.de/",
        endpointInterface = "de.sms.athena.service.RoutingOrderPortType",
        wsdlLocation = "routingOrderService.wsdl")
public class RoutingOrderServiceImpl extends BaseService
                        implements RoutingOrderPortType {
    ...
    @Override
    public AddRoutingOrdersResponse addRoutingOrders(final
        AddRoutingOrdersRequest request) {
        ...
    }
    ...
}
```

Quellcode 4.7: Ausschnitt der Klasse RoutingOrderServiceImpl

4.4 Update Schnittstelle

Im Gegensatz zur SOAP Schnittstelle werden über die Update Schnittstelle keine XML-basierten Nachrichten übertragen. Diese würden einen *Overhead* an Daten produzieren und den Update-Prozess deutlich verlangsamen.
Daher basiert die Update Schnittstelle auf einem einfachen zustandslosen Stream, über den die Daten übertragen werden.

Über die Schnittstelle werden Routing-Einträge von der Serverseite übertragen und auf der Clientseite als Routing-Aufträge gespeichert.

4.4.1 Aufbau der übertragenen Daten

1. `version`
   Gibt die Version der Update Schnittstelle als *Integer* an. Somit ist es möglich, dass zwei Athena Systeme mit verschiedenen Versionen die Schnittstelle nutzen. Dies geschieht, wenn zum Beispiel Änderungen am Protokoll durchgeführt wurden.

2. `tag`
   Ein *Integer* Wert, der angibt, welche Daten übertragen werden. Dadurch ist es in Zukunft möglich, die Update Schnittstelle zu erweitern.

3. `id`
   Gibt die Id des Routing-Eintrags an. Die zuletzt übertragene Id wird auf der *Client*-Seite am Update-Job gespeichert, damit die Übertragung beim nächsten Start des Jobs ab der Id fortgesetzt werden kann.

4. `rangeStart`
   Startnummer des Routings

5. `rangeEnd`
   Endnummer des Routings

6. `targetType`
   Typ des Routings

7. `targetValue`
   Routingziel

8. `switchDate`
   Schaltungsdatum des Routings

4.4.2 Performanceauswertung

Im Vergleich zur SOAP Schnittstelle wurde der Datendurchsatz in etwa verdoppelt. Im Schnitt werden circa drei Millisekunden pro Datensatz benötigt, was durch die Einsparung des XML-Aufbaus erreicht wurde.
Zwar hat diese Art der Übertragung auch Nachteile, wie zum Beispiel der festen Bindung an Java, diese sind in dem Kontext, in dem die Schnittstelle verwendet wird, jedoch weitestgehend irrelevant.

4.5 Anwendung eines Routing-Auftrags

Neue Routing-Informationen werden in Athena zuerst als Routing-Auftrag gespeichert und beeinflussen das Routing nicht. Erst im Zuge eines Verarbeitungsprozesses werden die Routing-Aufträge in Routing-Einträge überführt.
Dieser Prozess muss täglich mehrere zehntausend Routing-Aufträge anwenden, angrenzende Routing-Einträge zum selben Ziel zusammenführen und eventuell alte Routing-Aufträge zurück nehmen.
Die Performance ist auch hier wieder ein wichtiger Aspekt, denn erst wenn die erhaltenen Routing-Informationen als Routing-Einträge vorliegen, können der Konsolidierungs- und der Updateprozess gestartet werden.

4.5.1 Implementierung

Die folgende Abbildung 4.4 zeigt, welche Schritte bei jedem Routing-Auftrag durchgeführt werden müssen.

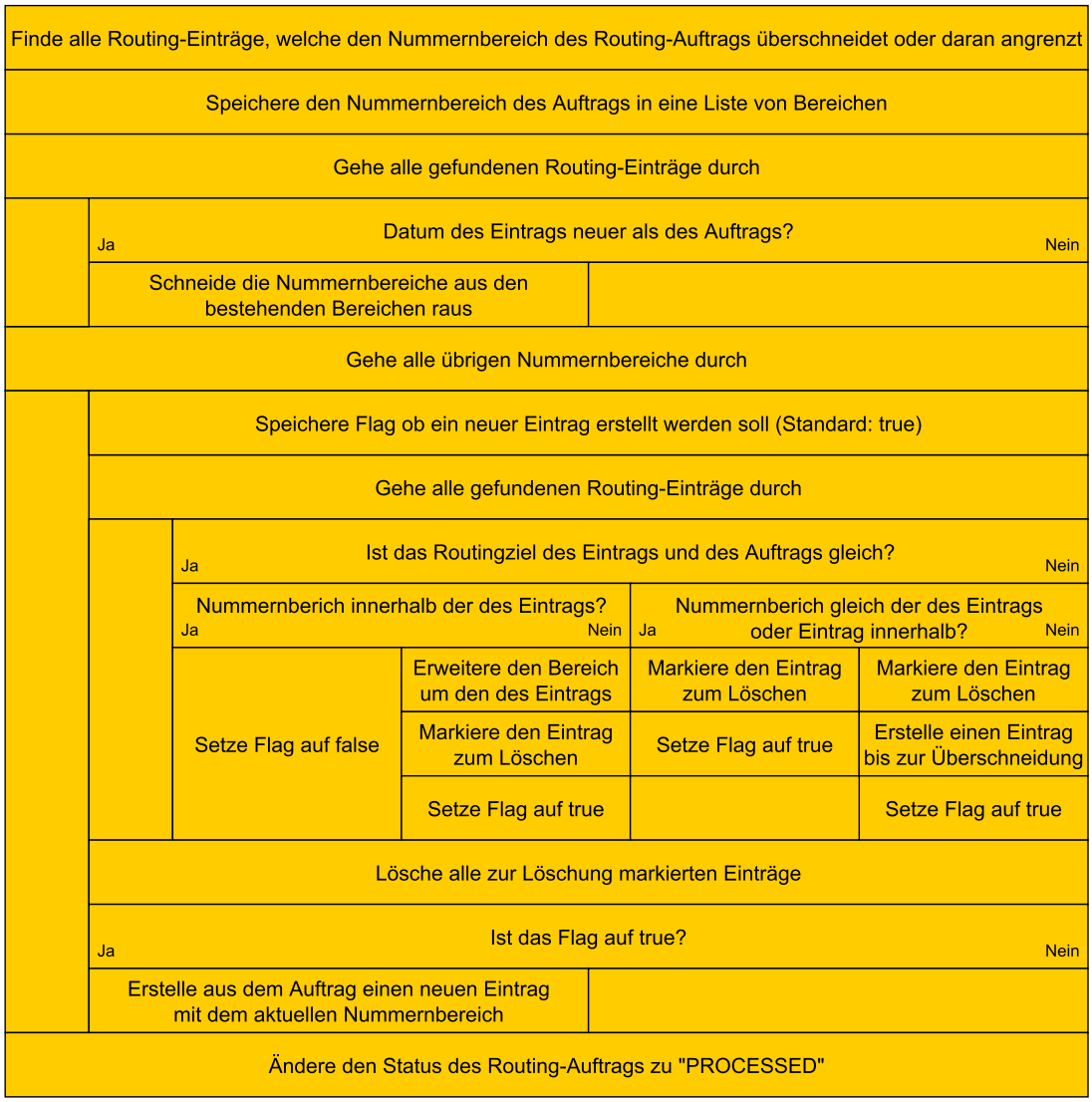

Abbildung 4.4: Nassi-Shneiderman-Diagramm zum Anwenden eines Routing-Auftrags

Zur Veranschaulichung des Prozesses dient die folgende Darstellung (Abbildung 4.5). In dieser wird ein Routing-Auftrag beispielhaft Schritt für Schritt verarbeitet und zum Schluss die erhaltenen Routing-Einträge dargestellt. Die Werte in den Klammern stellen das Schaltungsdatum vereinfacht dar.

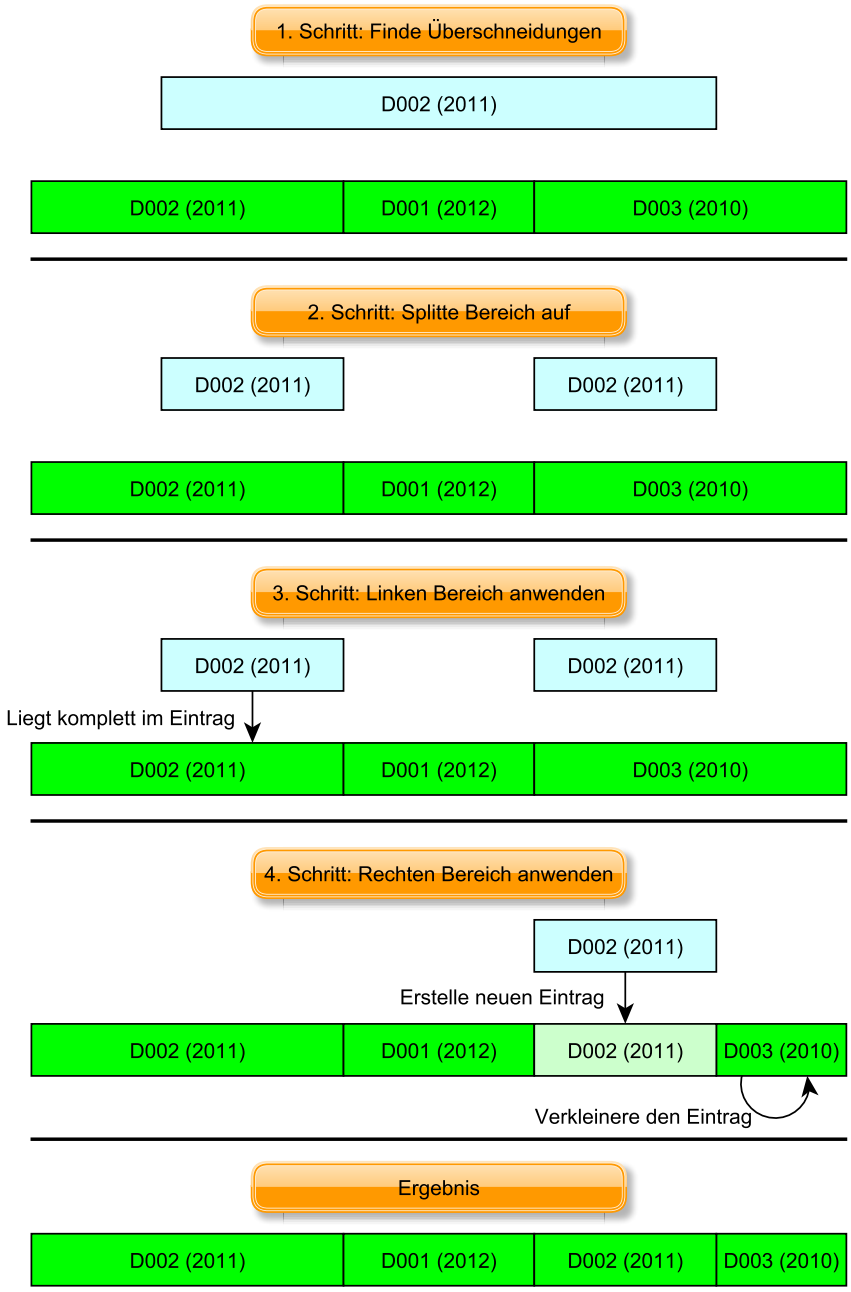

Abbildung 4.5: Beispielhafte Darstellung des Verarbeitungsprozesses eines Routing-Auftrags

4.5.2 Optimierung

Zur Zeit dieser Arbeit dauert die Anwendung eines Routing-Auftrags etwa sieben Millisekunden. Das vorgestellte Verfahren konnte nicht weiter optimiert werden, jedoch an

einer anderen Stelle.
Im ersten Entwurf verarbeitete der Prozess alle Routing-Aufträge innerhalb einer einzigen Datenbanktransaktion. Das führte zu immer länger laufenden SQL-Abfragen, da Hibernate innerhalb einer Transaktion viele Änderungen zwischenspeichert und bei Datenbankanfragen dann die Ergebnisse mit dem Speicher abgleichen muss.
Die Verarbeitung wurde im Verlaufe des Prozesses immer langsamer und Zeiten jenseits von 500 Millisekunden pro Routing-Auftrag entstanden.

Durch die Aufteilung der Prozedur in mehrere Transaktionen und dem regelmäßigen Leeren des Caches, wurde die Performance auf die obigen sieben Millisekunden stabilisiert.

```
@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
public int processPendingRoutingOrdersForTable(final Long tableId,
                                               final int quantity) {
    ...
    entityManager.flush();
    entityManager.clear();
}
```

Quellcode 4.8: Aufteilung in mehrere Transaktionen und Leeren des Caches

Die dafür notwendigen Änderungen sind sehr gering, wie der Quellcodeausschnitt 4.8 zeigt.
Die Annotation `@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)` gibt an, dass die Funktion in einer eigenen Transaktion ausgeführt wird.
Mit den Befehlen `flush()` und `clear()` werden zunächst alle zwischengespeicherten Daten an die Datenbank gesendet und anschließend der Speicher geleert.

4.6 Konsolidierung

Jede Routing-Gruppe wird in eine Routing-Tabelle zusammengefasst. Diese Konsolidierung verwendet die Logik zur Anwendung eines Routing-Auftrags (siehe Abbildung 4.4), um die zusammengefassten Informationen in die Routing-Tabelle zu schreiben.

Der Konsolidierungsprozess iteriert über alle Elemente einer Gruppe und bestimmt anhand dieser die neuen Routing-Einträge. Trifft er auf eine Subgruppe, wertet er zunächst diese aus und behandelt das Ergebnis im weiteren Prozess als eine temporäre Routing-Tabelle.
Je nach Verarbeitungstyp können sehr unterschiedliche Ergebnisse entstehen. Unterschieden wird zwischen den Typen *Priorität* und *Datum*. Die Verarbeitung nach Priorität ist immer die schnellere, da das erste gefundene Ergebnis auch direkt in die Routing-Tabelle geschrieben wird. Anders bei der Verarbeitung nach Datum, für die über alle Routing-Tabellen iteriert werden muss, um die neuste Routing-Informationen zu erhalten.

Die folgenden zwei Abbildungen zeigen eine beispielhafte Konsolidierung einer Routing-Gruppe mit drei Routing-Tabellen. Die erste Abbildung zeigt die Verarbeitung nach Priorität, die zweite nach Datum.

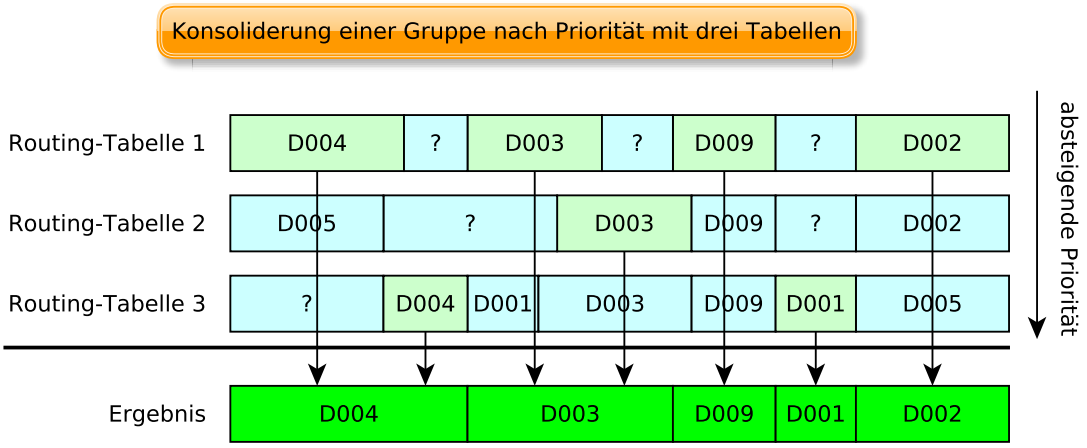

Abbildung 4.6: Beispielhafte Darstellung des Konsolidierungsprozesses nach Priorität

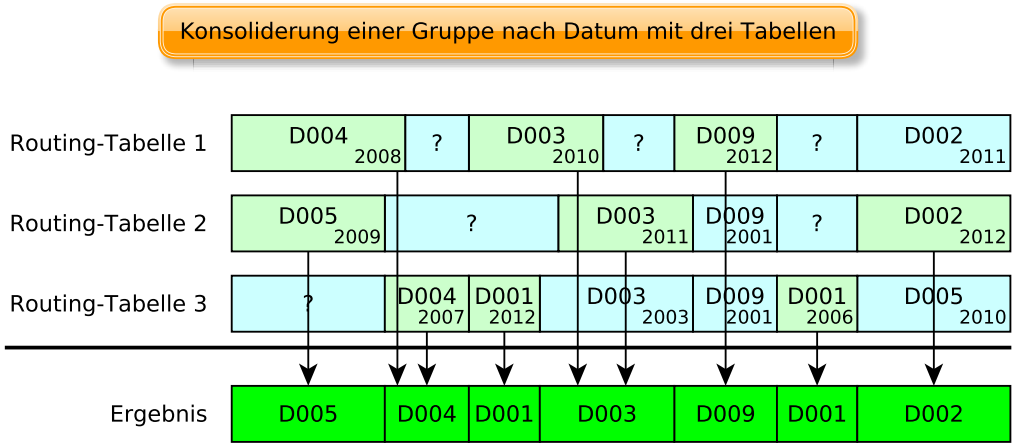

Abbildung 4.7: Beispielhafte Darstellung des Konsolidierungsprozesses nach Datum

Die Routing-Einträge der Routing-Tabellen sind in hellblau dargestellt, bzw. in hellgrün, wenn sie für das Ergebnis relevant sind. Die erstellten Routing-Einträge in der Ergebnistabelle sind grün dargestellt.
Der Konsolidierungsprozess benötigt etwas mehr Zeit pro Datensatz, als der Verarbeitungsprozess. Diese Zeitdifferenz entsteht durch die Iteration über die Routing-Tabellen und das Finden des richtigen Routings und ist abhängig von der Anzahl der Einträge einer Routing-Gruppe und dem Verarbeitungstyp.

Im Schnitt ist der Prozess etwa ein bis zwei Millisekunden pro Datensatz langsamer.

4.7 Test-Framework

Bevor eine Software an die Kunden ausgeliefert wird oder irgendwo zum Einsatz kommt, sollte sie vorher umfangreichen Tests unterzogen werden. Dazu zählen unter anderem Komponententests, Integrationstests, Systemtests und Abnahmetests. Häufig werden die Komponententests und auch die Integrationstests automatisiert, als so genannte *Unit-tests* durchgeführt.
In Java gibt es dafür die *JUnit*-Tests. Mit deren Hilfe, können die Entwickler bestimmte Funktionen der Software testen. Ein JUnit Test ist üblicherweise so aufgebaut, dass er eine Funktion mit bestimmten Daten aufruft und das Ergebnis gegen ein Soll-Ergebnis prüft. Werden Fehler geworfen oder ist ein Teil des Soll-Ergebnisses nicht erreicht worden, so ist der Test fehlgeschlagen.

JUnit-Tests kommen in Athena ebenfalls zum Einsatz. Mit ihnen wurde die allgemeine Funktionstüchtigkeit von Athena sichergestellt, sowie auch komplexere Abläufe simuliert. Zur Ausführung der Tests benötigt es einen *Embedded*-GlassFish, welcher ohne feste Installation im Speicher ausgeführt werden kann und außerdem eine eigene In-Memory Datenbank besitzt. Die Anwendung wird in diesen GlassFish eingebunden und anschließend die *JUnit*-Tests ausgeführt.

Der folgende Ausschnitt zeigt einen *JUnit*-Test, welcher die korrekte Funktionsweise der Funktion `isInside(NumberRange, NumberRange)` überprüft.

```
@Test
public void testIsInside() {
    assertTrue(Util.isInside(new NumberRange(5L, 7L, null),
                             new NumberRange(3L, 10L, null)));
    assertTrue(Util.isInside(new NumberRange(3L, 10L, null),
                             new NumberRange(3L, 10L, null)));
    assertTrue(Util.isInside(new NumberRange(4L, 5L, null),
                             new NumberRange(4L, 6L, null)));
    assertTrue(Util.isInside(new NumberRange(4L, 5L, null),
                             new NumberRange(3L, 5L, null)));

    assertFalse(Util.isInside(new NumberRange(1L, 10L, null),
                             new NumberRange(3L, 10L, null)));
    assertFalse(Util.isInside(new NumberRange(1L, 15L, null),
                             new NumberRange(3L, 10L, null)));
}
```

Quellcode 4.9: Ausschnitt eines JUnit-Tests

Die Tests werden automatisch bei jeder Änderung auf dem Entwicklungsserver der sms eSolutions GmbH durchgeführt und ausgewertet. Zusätzlich kann jeder Entwickler die Tests bei sich lokal auf dem Computer ausführen, um seine Änderungen zu überprüfen.

5 Zusammenfassung

5.1 Fazit

In dieser Arbeit wurden einige Teile der Entwicklung und Optimierung der Software Athena der sms eSolutions GmbH vorgestellt. Die Software befindet sich zur Zeit dieser Arbeit kurz vor dem geplanten Produktionsstart. Auf dem Server der sms eSolutions GmbH ist sie bereits in die Abläufe eingebunden und erhält die täglichen Routing-Informationen aus zwei Systemen. Bei einem Pilotkunden wurde sie ebenfalls installiert, wird zur Zeit von diesem aber noch getestet und konfiguriert.

Durch die in dieser Arbeit vorgestellten Lösungen und Optimierungen ist eine Software entstanden, welche von vielen verschiedenen Softwaresystemen über eine SOAP Schnittstelle mit Daten versorgt werden kann und diese komprimiert in einer Datenbank speichert.
Dabei verfügt Athena über mehr Funktionalitäten, als im Rahmen dieser Arbeit vorgestellt wurden; die hier vorgestellten Funktionen sind oftmals vereinfacht dargestellt oder zeigen einen Ausschnitt des Funktionsumfangs.

5.2 Ausblick

Im Laufe der nächsten Monate nach dieser Arbeit wird sich zeigen, ob Athena den hohen Anforderungen an Performance und Datenqualität gerecht werden kann. Dabei spielen die erhaltenen Daten aus den Fremdsystemen eine erhebliche Rolle, da Athena diese nur verwaltet, optimiert und zum Routing exportiert.
Übertragen die Systeme keine vernünftigen Daten oder werden die Prozesse in Athena langsamer, so müssen entweder die Fremdsysteme angepasst werden, oder am Athena weitere Optimierungen durchgeführt werden.

Bei den Kunden vor Ort wird sich zeigen, ob Athena das Routing in der Praxis verbessern und die Transit-Gebühren tatsächlich verringern kann.

Anhang

A.1 Das ER-Diagramm von Athena

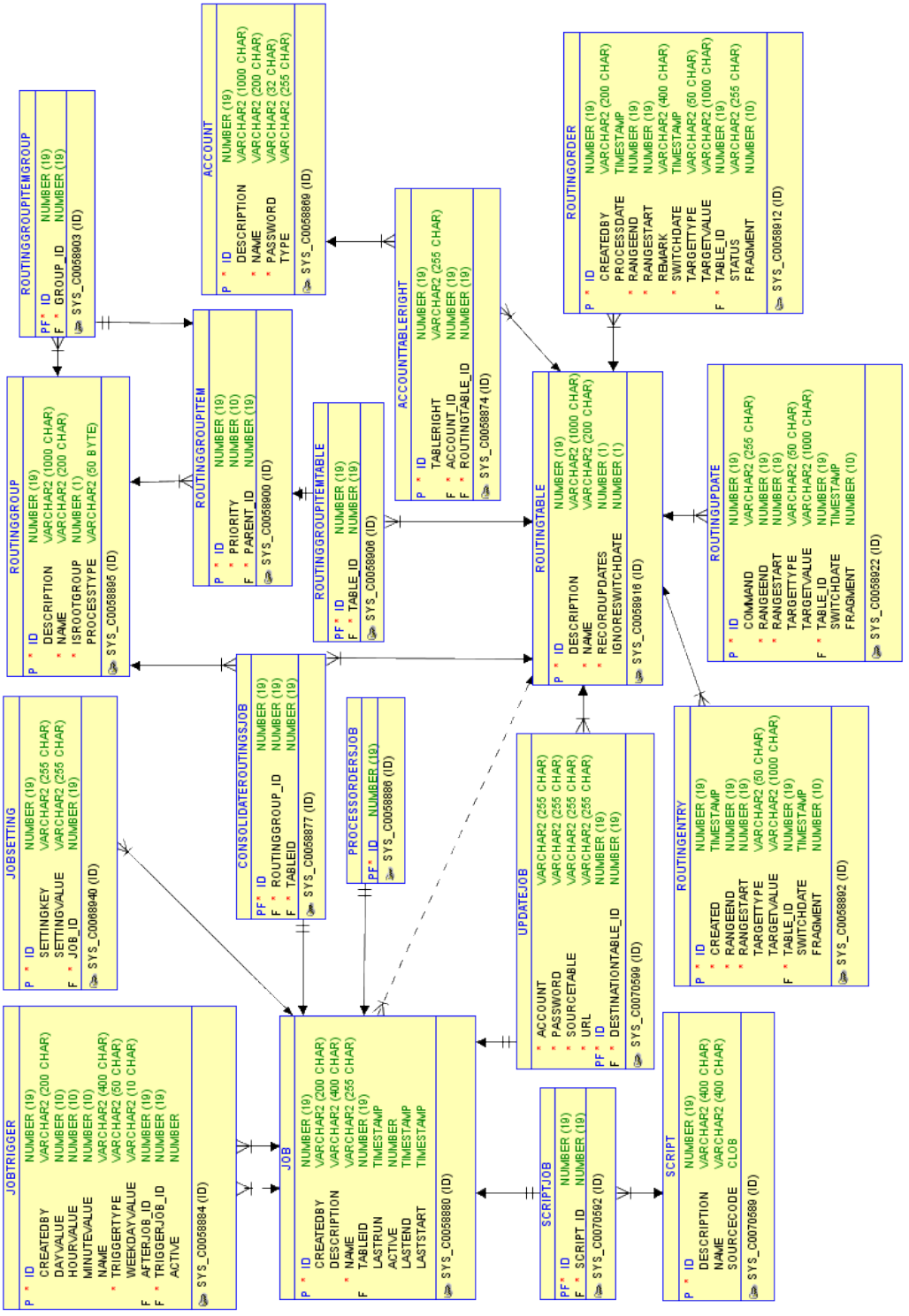

Abbildung A.1: ER-Diagramm von Athena

Glossar

Managed Bean
Eine Managed Bean ist eine von dem FacesServlet veraltete Java Klasse. Graphische Komponenten sind an die Eigenschaften dieser Beans gebunden und können Methoden an ihnen aufrufen.

Switch
Ein Switch ist eine Vermittlungsstelle für Telefonate. Bei einem Anruf muss diese innerhalb weniger Millisekunden den Anruf an den korrekten Teilnehmernetzbetreiber weiterleiten.

Transit-Anbieter
Ein Transit-Anbieter ist ein Teilnehmernetzbetreiber, welcher für andere TNBs das Routing übernimmt, wenn diese den Anruf einem anderen TNB nicht zuordnen können.

Abkürzungsverzeichnis

BNetzA	Bundesnetzagentur.
CDI	Context and Dependency Injection.
DBMS	Datenbankmanagementsystem.
EJB	Enterprise JavaBeans.
GUI	Graphical User Interface.
HTML	Hypertext Markup Language.
HTTP	Hypertext Transfer Protocol.
ICCS	Inter Carrier Communication System.
JAX-WS	Java API for XML Web Services.
JDBC	Java Database Connectivity.
JEE	Java EE.
JMS	Java Message Service.
JPA	Java Persistence API.
JSF	JavaServer Faces.
JSP	JavaServer Pages.
JSR	Java Specification Request.

JTA	Java Transaction API.
ORM	Object-Relational Mapping.
POJO	Plain Old Java Object.
RI	Referenzimplementierung (engl. *Reference Implementation*).
SOAP	Simple Object Access Protocol.
SQL	Structured Query Language.
TNB	Teilnehmernetzbetreiber.
WSDL	Web Services Description Language.
XML	Extensible Markup Language.
XSD	XML Schema Definition.

Literaturverzeichnis

[Wikipedia] Wikipedia, Stichwort „Java Database Connectivity“, Version vom 2. Mai 2012, abrufbar unter `http://de.wikipedia.org/w/index.php?title=Java_Database_Connectivity&oldid=102731747` (abgerufen am 15.07.2012)

[Wikipedia] Wikipedia, Stichwort „Session Beans“, Version vom 27. April 2012, abrufbar unter `http://en.wikipedia.org/w/index.php?title=Session_Beans&oldid=489490536` (abgerufen am 16.07.2012)

[Wikipedia] Wikipedia, Stichwort „Datenbankindex“, Version vom 23. Mai 2012, abrufbar unter `http://de.wikipedia.org/w/index.php?title=Datenbankindex&oldid=103561660` (abgerufen am 04.08.2012)

[Open-Source-Tag Magdeburg] Open-Source-Tag Magdeburg, PDF von Andreas Scherbaum, abrufbar unter `http://andreas.scherbaum.la/writings/open-source-tag-md-2008-10_talk.pdf` (abgerufen am 16.07.2012)

[Beginning Java EE 6] Antonio Goncalves (2010): Beginning Java™ EE 6 Platform with GlassFish™ 3, From Novice to Professional, apress Verlag, Second Edition

[w3schools] w3schools, Stichwort: „SOAP“, `http://www.w3schools.com/soap/default.asp` (abgerufen am 17.07.2012)

[it-republik] it republik, Stichwort: „Java EE6 auf einen Blick“, `http://it-republik.de/jaxenter/artikel/Java-EE-6-auf-einen-Blick-2759.html` (abgerufen am 23.07.2012)

[teltarif] teltarif, Stichwort: „Regulierung senkt Festnetz-Zusammenschaltungskosten deutlich“, `http://www.teltarif.de/festnetz-interconnect-terminierung-bundesnetzagentur/news/43193.html` (abgerufen am 31.07.2012)

[TecChannel] TecChannel, Stichwort: „SQL-Optimierung: Indizes richtig einsetzen“, `http://www.tecchannel.de/server/sql/1753761/sql_optimierung_indizes_richtig_einsetzen/index8.html` (abgerufen am 05.08.2012)