h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

DFS Deutsche Flugsicherung

Hochschule Darmstadt

– Fachbereich Informatik –

in Kooperation mit

DFS Deutsche Flugsicherung GmbH

– Lage- und Informationszentrum – OA/L –

Erweiterung und Integration des Backend-Processing von Radardaten in den Enterprise-Service-Bus EDIS

Abschlussarbeit zur Erlangung des
akademischen Grades Bachelor of Science (B.Sc.)

vorgelegt von

Philipp Gerome

Matrikelnummer : 735236

Referent : Prof. Dr. Oliver Weissmann

Korreferent : Prof. Dr. Hans-Peter Wiedling

Fachbetreuer : Prof. Dr. Kai Renz

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Langen, den 08. März 2017

Sperrvermerk

Die vorliegende Arbeit mit dem Titel:

Erweiterung und Integration des Backend-Processing von Radardaten in den Enterprise-Service-Bus EDIS

beinhaltet interne, vertrauliche Informationen der Firma:

DFS Deutsche Flugsicherung GmbH.

Die Weitergabe des Inhalts der Arbeit sowie die Zeichnungen und Daten, im Gesamten oder in Teilen ist grundsätzlich untersagt. Es dürfen keinerlei Kopien oder Abschriften - auch in digitaler Form - gefertigt werden.

Ausnahmen bedürfen der schriftlichen Genehmigung durch:

DFS Deutsche Flugsicherung GmbH
Am DFS Campus 10
63225 Langen

Langen, den 08. März 2017

Kurzfassung

Das Lage- und Informationszentrum (kurz LIZ) der DFS Deutschen Flugsicherung GmbH erstellt täglich für rund 10.000 Flüge und daraus resultierend 6.000.000 Radarkoordinaten Statistiken. Für die Zuweisung der Radarkoordinaten zu einem Sektor wurde in einer vorhergehenden Arbeit eine Octreestruktur entwickelt. Diese ermöglicht eine Zuweisung der Koordinaten zu den jeweils passenden Sektoren eines kompletten Tages in 900 ms[1].

Die bisherige Erzeugung eines Octrees für die Sektorkarte der DFS dauerte 40 Sekunden. Diese wird in der vorliegenden Arbeit durch eine effizientere Methode erweitert und ersetzt. Jene Methode verringert die Aufbauzeit um 82 % auf sieben Sekunden. Zusätzlich wurde für die Speicherung eines Octree-Exportes eine Datenbanklösung mit Oracle MySQL entwickelt. Um die Transaktionszeiten zu verringern, wird der Octree-Export in der Datenbank nur komprimiert abgespeichert. Ergänzend dazu wurde ein Statistikmodul entwickelt. Dieses erstellt Statistiken zur Sektorauslastung sowie zu Flugevents. Dabei wird ebenfalls die zurückgelegte Strecke jedes Flugzeuges innerhalb des Luftraumes berechnet. Die erstellten Statistiken werden mit Hilfe von Apache OpenJPA ebenfalls in einer Oracle MySQL-Datenbank gespeichert.

Beide Module werden mit Apache Felix in OSGi-Bundles überführt. Diese sollen anschließend in EDIS integriert werden. EDIS ist der Enterprise-Service-Bus innerhalb des LIZ, der unter anderem alle Radardaten an verschiedene Systeme verteilt. Die verteilten Radardaten sollen durch den Octree dem passenden Sektor zugewiesen werden. Das Statistikmodul soll später, innerhalb von EDIS, Statistiken zu den live eintreffenden Radardaten erstellen.

[1] Alle Zeitmessungen wurden auf einem System mit einem i7 3770K @ 4x3.50 GHz und 16 GB DDR3 RAM durchgeführt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

Die DFS Deutsche Flugsicherung GmbH (kurz DFS) ist ein privatrechtlich organisiertes Unternehmen im Besitz der Bundesrepublik Deutschland. Sie ist für die Flugverkehrskontrolle des deutschen Luftraumes zuständig und kontrolliert hierbei im Jahr etwa drei Millionen Flugbewegungen [dfs16]. Für die Kontrolle von An- und Abflügen, sowie des Flughafennahbereichs, sind die 16 Tower-Standorte an den internationalen Flughäfen in Deutschland verantwortlich. Die darüber hinausgehende Streckenkontrolle erfolgt in den vier Area-Control-Zentralen in Deutschland [dfs17].

Der deutsche Luftraum ist in Sektoren organisiert. Abbildung 1.1 zeigt eine vereinfachte Darstellung dieser Sektorstruktur. Das Lage- und Informationszentrum (kurz LIZ) der DFS muss täglich Statistiken zu rund 10.000 Flügen erstellen. Über den Tag verteilt treffen im LIZ dazu circa 6.000.000 Radarpositionen ein. Statistiken werden sowohl für Sektoren als auch für Flüge erstellt.

Bisher werden die Statistiken im LIZ mit dem STANLY 3 System erzeugt. STANLY 3 (im Folgenden als STANLY bezeichnet) wird von einem externen Anbieter entwickelt. Das Produkt soll durch ein DFS-eigenes System ersetzt werden, da dieses in der Weiterentwicklung, Unterhaltung und Wartung weniger kostenintensiv ist [LIZ17].

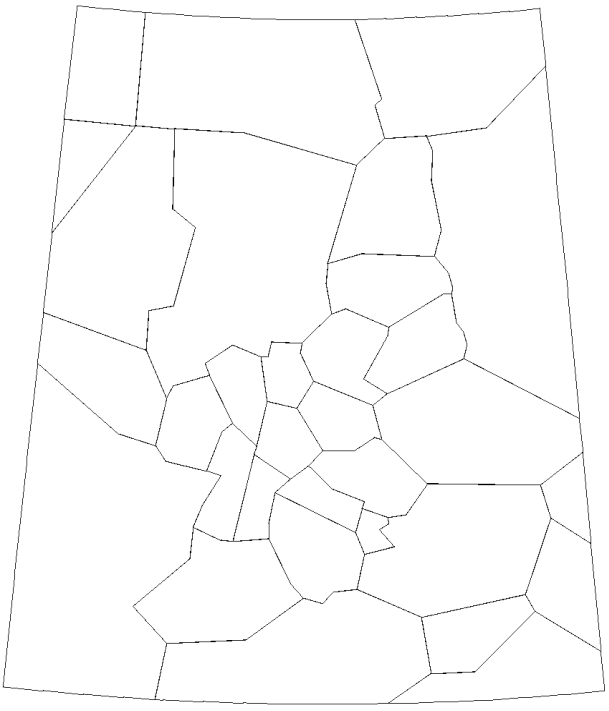


Abbildung 1.1: *Vereinfachte Sektorstruktur über Deutschland*

1.2 Problemstellung und Zielsetzung

Für das Zuordnen der Radarpositionen zu einem Sektor wurde bereits eine Octreestruktur entwickelt [Ger16]. Die Erzeugung dieser Struktur benötigt mindestens 40 Sekunden. Die vorhandenen Methoden zur Erzeugung sollen optimiert werden. Dies soll zukünftig zu einer Minimierung der Aufbauzeiten führen. Für eine schrittweise Reduzierung der Abhängigkeit von STANLY, soll der Octree-Suchalgorithmus anschließend um ein Statistikmodul erweitert werden. Dieses soll vorerst analog zum Altsystem Statistiken zur Verfügung stellen. Zudem sollen die Statistiken um einige Features erweitert werden. Hierzu zählt z. B. die in einem Sektor zurückgelegte Strecke. Des Weiteren sollen Sektor Ein- und Austrittspunkte mit einer größeren Genauigkeit bestimmt werden. Die erstellten Statistiken sollen persistent in einer Datenbank gespeichert und für weitere Berechnungen verwendet werden können. Die optimierte Octree-Struktur sowie das Statistikmodul sollen prototypisch mittels OSGi-Framework in ein oder mehrere Bundles überführt werden. Um die Funktionen dieser Bundles zu überprüfen, sollen sie in eine Testinstanz von EDIS integriert werden.

1.3 Aufbau der Bachelorarbeit

Zunächst wird in diesem Dokument auf die bereits in einer vorhergehenden Arbeit [Ger16] behandelte Octreestruktur eingegangen. Hierbei werden im Speziellen weitere Optimierungen sowie die Erweiterung um eine Factory erläutert. Anschließend wird auf das Statistikmodul eingegangen. Hier werden die Anforderungen an die Software formuliert und analysiert. Darüber hinaus werden die Implementierung und das Testverfahren beschrieben. Im darauffolgenden Kapitel wird die Integration des Statistikmoduls in die Umgebung von EDIS dargestellt. Abschließend wird das Dokument mit einem Fazit und einem Ausblick in die Zukunft des Softwareproduktes beendet.

2 Grundlagen der vorliegenden Arbeit

In diesem Kapitel wird die Sektorstruktur des deutschen Luftraumes beleuchtet. Anschließend wird auf die bereits entwickelte Octree-Struktur eingegangen. Zudem wird das Umfeld beschrieben, in das der Octree und das Statistikmodul später integriert werden.

2.1 Luftraumstrukturen

Der deutsche Luftraum ist in 133 einzelne Sektoren unterteilt. Die Unterteilung kann Abbildung 2.1 entnommen werden. Sektoren selbst bestehen ebenfalls aus kleineren Bestandteilen. Diese kleineren Lufträume werden als Airblocks bezeichnet [Ger16, S. 4 f.]. Airblocks besitzen eine polygonale Grundfläche und eine feste Start- und Endhöhe. Luftfahrzeuge innerhalb eines Sektors werden von je zwei Fluglotsen betreut. Nach Möglichkeit können verkehrsschwache Sektoren zu einer Konfiguration zusammenlegt werden. Diese Konfiguration wird ebenfalls durch zwei Lotsen betreut. Für die bessere Planung der Flugverkehrskontrolle spielt die Erkennung von Engpässen, Freiräumen und überdurchschnittlicher Auslastung eine essenzielle Rolle. Hierzu ist es notwendig, Statistiken über die Auslastung von Sektoren und Konfigurationen zu führen.

Abbildung 2.1: *Zweidimensionale Darstellung der Sektoren und Airblocks des deutschen Luftraumes*

Damit Flüge innerhalb Deutschlands auch zu einem späteren Zeitpunkt nachverfolgt werden können, ist es notwendig verschiedene Events zu speichern. Hierzu zählen beispielsweise Sektorein- und Sektoraustritte.

2.2 Verwendung eines Octrees

In der vorhergehenden Praxisarbeit [Ger16] wurde eine Octree-Datenstruktur entwickelt. Diese Struktur zerteilt eine Sektorkarte rekursiv in acht gleich große Würfel. Ziel dieser Unterteilung

ist das Erzeugen eines Suchbaumes mit eindeutigen Blattelementen. Innerhalb des erstellten Suchbaumes können alle Radarkoordinaten eines Tages (circa 6.000.000) in 900 ms[1] einem Sektor zugeordnet werden. Die Erzeugung dieses Baumes benötigt ca. 40 Sekunden.

2.3 Umfeld der Anwendung

Das entwickelte Statistikmodul soll in den Enterpise-Service-Bus EDIS des LIZ integriert werden. Über EDIS werden verschiedene Systeme und Abteilungen der DFS mit den Radar- und Flugplandaten des aktuellen Flugverkehrs des deutschen Luftraumes versorgt. Das Statistikmodul soll innerhalb von EDIS integriert werden.
Den Kern von EDIS bildet Apache Karaf. Karaf ist ein Container, der verschiedene OSGi-Services und Bundles installieren und zur Laufzeit starten kann [HK14]. Dies führt zu einer modularen Struktur innerhalb von EDIS. Jegliche Module können innerhalb von Karaf installiert und von anderen Modulen verwendet werden [apa16]. Innerhalb von EDIS läuft ein Modul mit einer Apache Camel Implementierung. Camel ist eine Java-basierte Routing-Engine [apa15]. Diese übernimmt das Routing für eintreffende Daten und verteilt diese an verschiedene Systeme der DFS sowie an verschiedene Module von EDIS.
Berechnete Daten und Statistiken sollen in Datenbanken innerhalb des LIZ-Netzwerkes gespeichert werden. Module, die in den EDIS-Container integriert werden, müssen in Java implementiert werden. Innerhalb der LIZ Infrastruktur wird der Aufbau des deutschen Luftraumes in den zwei verschiedenen Formaten der Systeme STANLY und FRODO gespeichert. FRODO ist ein System zur Darstellung von Lufträumen und Sektoren.

[1] Alle Zeitmessungen wurden auf einem System mit einem i7 3770K @ 4x3.50 GHz und 16 GB DDR3 RAM durchgeführt.

3 Optimierung der bestehenden Octree-Struktur

In diesem Kapitel wird die Optimierung des bestehenden Octrees zur Sektorsuche beschrieben. Wie bereits in der vorhergehenden Arbeit vermerkt, wird die Erzeugung des Octrees verkürzt [Ger16, S. 25]. Darüber hinaus wird der Octree um eine Octreefactory erweitert. Diese soll Octrees verwalten und den aktuell gültigen Octree zur Verfügung stellen.

3.1 Laufzeitoptimierung

Die Erstellung eines Octrees für eine Sektorkarte dauert ca. 40 Sekunden. Im Falle eines Neustarts der Anwendung können die einlaufenden Radardaten nicht ausgewertet werden, bis der Octree zum Auswerten erstellt worden ist. Dadurch müssen alle Daten, die innerhalb der Aufbauzeit eintreffen, zwischengespeichert werden. Deshalb soll die Erzeugung des Octrees weiter optimiert werden. Laufzeitmessungen zeigen, dass ca. 98,99 % der benötigten Laufzeit für die Berechnung der Sektoren zu einem Octreeelement verstreichen. Hierbei wird für jedes Octreeelement jeder verfügbare Sektor mittels Polygonsubtraktion geprüft. Hier wird die Grundfläche des Sektors von der des Octreeelementes subtrahiert. Verändert sich hierbei die Fläche

des subtrahierten Objektes, schneiden sich beide Flächen. Um das Aufbauverfahren des Baumes zu verkürzen muss diese Berechnung analysiert und verbessert werden. Das Vorgehen wird zur besseren Veranschaulichung zweidimensional betrachtet und dargestellt. Nichtsdestotrotz wird im Folgenden die Bezeichnung Octree beibehalten, obwohl es sich hierbei korrekterweise um einen Quadtree handelt. In der dreidimensionalen Betrachtung werden die Berechnungen um eine Höhe erweitert. Die Höhen eines Octreeelementes und eines Airblocks sind fest. Die Höhe ist an jedem Punkt der jeweiligen Grundfläche der Elemente identisch. Das Prüfen eines Schnittes in der Höhe ist folglich im Vergleich zu der Fläche weniger aufwendig.

3.1.1 Möglichkeiten einer Schnittprüfung zwischen Polygonen

In der bisherigen Implementierung wird geprüft ob ein Sektor in einem Octreeelement[2] liegt indem die Grundfläche des Sektors, von der des Octreeelementes subtrahiert wird (Polygonsubtraktion). Verkleinert sich hierbei die Fläche des Octreeelementes, schneiden sich die beiden Objekte. Sollte die verkleinerte Fläche des Octreeelementes nach der Subtraktion leer sein, füllt der Sektor das Octreeelement vollständig aus. Hier liegt exklusiv nur der geprüfte Sektor im Octreeelement und damit das Octreeelement vollständig im Sektor. Liegt bei einer Suche die Koordinate in einem Octreeelement mit nur einem Sektor, ist innerhalb des Octreeelementes keine weitere Sektorprüfung notwendig[3]. Die verwendeten Polygonflächen werden in Objekten des Typs java.awt.geom.Area abgebildet. In einem Laufzeitvergleich wurden drei Methoden getestet. Hier wurden Polygone mit einer Anzahl von n-Ecken mit je 2.000.000 Rechtecken – welche die Polygone schneiden – geschnitten. Bei den Polygonen und Rechtecken handelt es sich nicht um Beispiele aus dem Luftraum, sondern um zufällig erzeugte Flächen.

[2] korrekterweise Quadtreeelement

[3] Unter der Annahme, dass sich Sektoren nicht selbst überlappen

Anzahl Ecken	Dauer in [ms]		
	Polygonsubtraktion	Intersect Methode	Boundingbox
4	7.731	126	18
8	7.141	159	18
16	11.475	197	17
32	17.438	311	17
64	30.706	501	18
128	54.633	920	17

Tabelle 3.1: *Versuchsdauer in Millisekunden bei Rechtecken, die das Polygon schneiden*

Tabelle 3.1 zeigt die Ergebnisse der Laufzeittests. Wie diesen entnommen werden kann, ist die Subtraktion der Polygone von den Octree-Flächen im Vergleich zu den anderen Methoden sehr zeitintensiv. Bei jeder Subtraktion wird zunächst die Ergebnisfläche als neues Polygon erstellt [ora16]. Erst anschließend kann geprüft werden, ob das Subtraktionsergebnis verändert oder leer ist. Das Prüfen, ob ein Rechteck in einem Sektor (128-Eck) liegt, dauert mit dieser Methode für 2.000.000 Rechtecke über 50 Sekunden. Die reale Datengrundlage besteht aus 133 Sektoren (1.086 Polygone) und 3,1 Millionen Octreeelementen [Ger16, S. 5]. Die Polygone verfügen im Schnitt über acht Eckkoordinaten. Im Vergleich zur verwendeten Testumgebung, werden beim realen Aufbau eines Octrees mehr Schnitte geprüft. Die Polygone und Rechtecke, die beim realen Erzeugen eines Octrees verwendet werden, sind jedoch weniger komplex. Die Laufzeit dieser Testumgebung ist mit 54 Sekunden etwas höher, als die der tatsächlichen Erzeugung mit 40 Sekunden.

Bei der Untersuchung des Funktionsumfanges der Libary java.awt.geom, fällt auf, dass es alternative Methoden mit der gewünschten Funktionsweise gibt. Hierbei erhalten die Funktionen Area.intersects(Rectangle2D) und Area.contains(Rectangle2D) besondere Bedeutung. Beim Vergleichen der Laufzeiten der intersect-Methode (Tabelle 3.1 3. Spalte) und der Polygonsubtraktion fällt auf, dass die Laufzeiten bei identischen geometrischen Formen deutlich unter denen der Polygonsubtraktion liegen. Bei einem ebenfalls 128-eckigen Polygon und identischen Rechtecken benötigt die Intersect-Methode nur 920 ms. Die Intersect-Methode prüft nur, ob der geprüfte Sektor das Octreeelement schneidet. Hierbei betrachtet die Methode

beide Formen als Fläche und nicht ausschließlich ihre Rahmen. Ein Rechteck, das folglich komplett innerhalb eines Polygons liegt, schneidet das Polygon trotzdem. Schneidet der Sektor bei der Intersect-Methode das Octreeelement, liegt das Octreeelement unter anderem im entsprechenden Sektor. Octreeelemente, die nach der Prüfung aller Sektoren ausschließlich einen einzigen Sektor in sich haben, müssen über die Contains-Methode weiter betrachtet werden. Sie prüft hierbei, ob der Sektor das komplette Octreeelement füllt.

3.1.2 Anwenden der Möglichkeiten auf die Octree-Struktur

Die Contains-Methode wird auf dem Area-Objekt der Grundfläche eines Sektors ausgeführt. Hier wird die Grundfläche des Octreeelementes als Rectangle2D übergeben. Diese Herangehensweise liefert das selbe Ergebnis, führt jedoch deutlich schneller zu diesem als die Polygonsubtraktion. Da die Grundflächen von Sektoren konvexe Polygone sein können, ist es nicht ausreichend, ausschließlich zu prüfen, ob jede Eckkoordinate eines Octreeelementes im Sektor liegt. Dies ist in Abbildung 3.1(a) (Seite 14) verdeutlicht.
Es können weitere Laufzeiteinsparungen erzielt werden. Jede Area erhält in ihrem Konstruktor eine Boundingbox. Diese Boundingbox beschreibt das kleinste Rechteck um das Polygon, das alle Koordinaten eines Polygons beinhaltet [Hua12]. In der vierten Spalte der Tabelle 3.1 ist ebenfalls die Laufzeit dieser Methode aufgeführt. Hierbei ist erkennbar, dass die Laufzeiten nicht von der Anzahl der Ecken des Polygons abhängen. Das Verfahren benötigt bei identischer Testumgebung nur 18 ms um einen Schnitt der Boundingbox mit den rechteckigen Octreeelementen zu erkennen. Das Schneiden der Boundingboxen kann allerdings nicht verwendet werden, um zu prüfen ob ein Rechteck ein Polygon schneidet, dies ist in Abbildung 3.1(c) (Seite 14) verdeutlicht. Um mit den Boundingboxen einen Schnitt sicher festzustellen zu können, müsste die Fläche des Polygons in viele kleine eindeutige Vierecke – also in einen Quadtree – unterteilt werden [Ger16, S. 8 f.]. Alternativ dazu kann die Methode verwendet werden, um eine Vorauswahl zu treffen. Mithilfe der Boundingbox können diejenigen Sektoren ausgeschlossen werden, die definitiv nicht näher geprüft werden müssen. Hierbei liefert das

Boundingbox-Verfahren bei Sektoren, deren Boundingbox das Octreeelement nicht schneidet, das exakte Ergebnis bei kürzestester Laufzeit und damit die höchste Effizienz. Dies wird in der Arbeit [XLC16] bestätigt. Hat ein Sektor die Vorauswahl erfolgreich bestanden, muss er mit der Contains- und Intersect-Methode weiter geprüft werden.

In der vorhandenen Implementierung wird nun anstatt der Polygonsubtraktion die Intersect und die Contains-Methode der Area-Klasse verwendet. Bevor diese jedoch ausgeführt werden, wird zunächst eine Vorauswahl der Sektoren durch den Schnitt der Boundingboxen durchgeführt. Schneiden sich die beiden Rechtecke, werden beide Flächen näher betrachtet. Hierbei wird die Intersect-Methode auf der Area der Sektors ausgeführt. Übergeben wird der Methode die rechteckige Grundfläche des Octreeelementes. Fällt auch diese Prüfung positiv aus, muss über die Contains-Methode geprüft werden, ob die Grundfläche des Octreeelementes vollständig im Sektor liegt. Dies ist im folgenden Quellcode ohne Betrachtung der Höheninformationen dargestellt.

```
1  public class OctreeElement {
2      [...]
3      Rectangle2D flaeche;
4
5      public enum CutType {
6          CUT, INSIDE, NONE
7      }
8
9      /*
10     * Gibt zurück, ob der Sektor im OctreeElement (2D = Quadtree) liegt
11     * oder ihn sogar komplett füllt.
12     * Hierbei bedeutet der zurückgegebene Enum-Wert:
13     * INSIDE: In diesem OctreeElement, liegt ausschließlich der geprüfte Sektor
14     * CUT: Der geprüfte Sektor liegt zu einem Teil in dem OctreeElement.
15     * NONE: Der geprüfte Sektor liegt nicht in dem OctreeElement
16     */
17
18     public int checkSektor (Sektor sektor) {
19         Area area = sektor.getArea();
20         Rectangle2D boundingBox = area.getBounds2D();
21         //Boundingbox und OctreeElement schneiden sich?
22         if(boundingBox.intersects(this.flaeche)){
23             //OctreeElement und Grundfläche des Sektors schneiden sich?
24             if(area.intersects(this.flaeche)){
25                 //OctreeElement liegt komplett in der Grundfläche des Sektors
26                 if(area.contains(this.flaeche)){return CutType.INSIDE;};
27                 return CutType.CUT;
28             }
29         }
30         return CutType.NONE;
31     }
32 }
```

Quellcode 3.1: *Quellcode zur Überprüfung, ob ein Octreeelement in einem Sektor liegt*

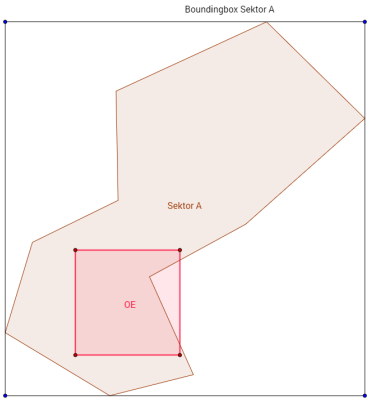
3.1.3 Validierung der dargestellten Polygonschnitt-Funktion

Im Folgenden wird die im vorherigen Abschnitt beschriebene Polygonschnitt-Funktion validiert. Hierzu wird die Laufzeit der Methode bei verschiedenen Testfällen bestimmt. Die resultierenden Ergebnisse werden anschließend mit einer Analyse der Octrees gewichtet.

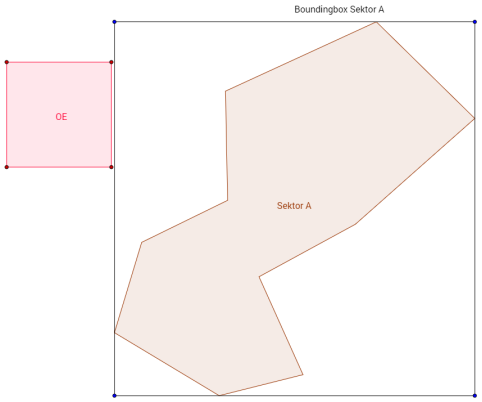
Laufzeitvergleich

In einer Validierung wird die ursprüngliche Polygonsubtraktion und die beschriebene neue Möglichkeit gegenübergestellt. Hierbei werden sowohl die Laufzeit, als auch die Ergebnisse betrachtet. Je Versuch wurden zwei Millionen Rechtecke mit einem zufälligen Test-Polygon mit n-Ecken geschnitten. Beim Schneiden der Flächen wird geprüft, ob das Rechteck das Polygon schneidet oder sogar vollständig in ihm liegt. Für die Versuche werden sechs verschiedene Mengen von Rechtecken gewählt.

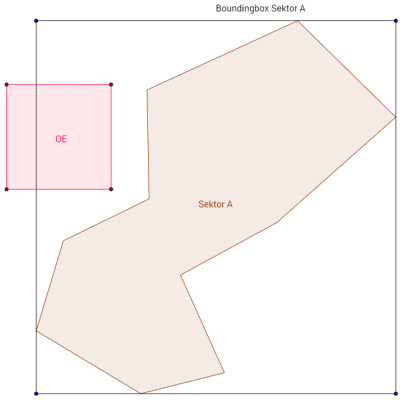
- Rechtecke, die außerhalb der Boundingbox liegen (Abbildung 3.1(b))
- Rechtecke, die vollständig im Polygon liegen (Abbildung 3.1(f))
- Rechtecke, die die Boundingbox, aber nicht das Polygon schneiden (Abbildung 3.1(c))
- Rechtecke, die in der Boundingbox liegen, aber das Polygon nur schneiden (Abbildung 3.1(e))
- Rechtecke, die die Boundingbox und das Polygon schneiden (Abbildung 3.1(d))
- Rechtecke, die auf einer Strecke durch das Polygon verteilt sind (alles aus Abbildung 3.1)



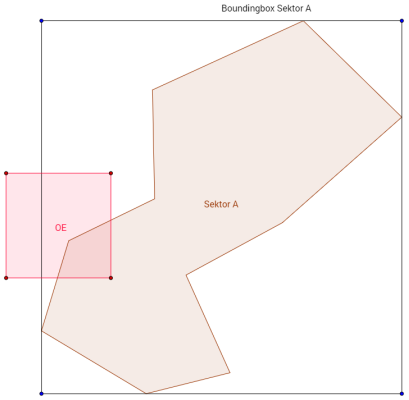
(a) Problematik einer Contains-Berechnung bei konvexen Polygonen



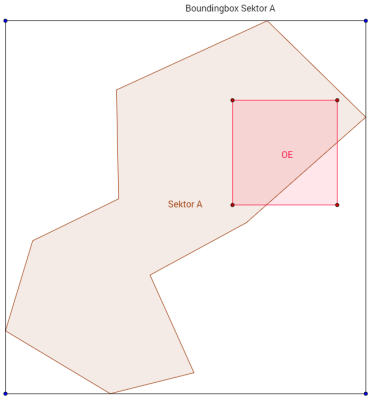
(b) Octreeelement außerhalb der Boundingbox



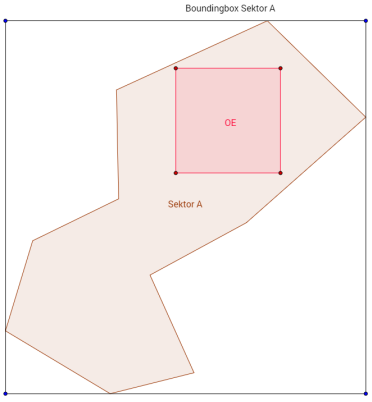
(c) Octreeelement schneidet die Boundingbox, nicht aber den Sektor



(d) Octreeelement schneidet die Boundingbox und den Sektor



(e) Octreeelement schneidet den Sektor innerhalb der Boundingbox



(f) Octreeelement vollständig im Sektor

Abbildung 3.1: *Beispielhafte Situationen bei der Beziehung zwischen Sektoren, Octreeelementen und der Boundingbox*

Im Folgenden werden die Ergebnisse der Tests dargestellt. Bei den getesteten Laufzeiten handelt es sich um Durchschnittswerte.

Anzahl Ecken	Dauer in [ms]	
	Polygonsubtraktion	getestete Methode
4	5.397	19
8	6.171	19
16	7.696	20
32	10.362	19
64	16.012	21
128	25.580	23

Tabelle 3.2: *Testergebnisse für Rechtecke, die außerhalb des Polygons liegen, in Millisekunden*

In dieser Versuchsreihe schneiden die Rechtecke weder das Polygon, noch die Boundingbox des Polygons. Die Laufzeiten wurden in Tabelle 3.2 dokumentiert. Dabei zeigt sich, dass mit einer zunehmenden Anzahl von Ecken des Polygons die Laufzeit der Polygonsubtraktion deutlich zunimmt. Dies kann durch die Implementierung hergeleitet werden und bestätigt die Tests aus Tabelle 3.1. Dabei muss für jeden Fall das Polygon vollständig subtrahiert werden. Alle Ecken des Polygons müssen in diese Berechnung eingeschlossen werden. Die Laufzeit der zu testenden Methode bleibt über die Anzahl der Ecken konstant. Durch die vorherige Boundingbox-Prüfung ist es nicht notwendig die Methoden Contains und Intersect auszuführen. Der Boundingboxtest erkennt, dass das Rechteck nicht in der Boundingbox liegt und führt so auch keine weiteren Berechnungen durch. Das Schneiden der Boundingbox mit dem Rechteck ist nicht von der Anzahl der Ecken abhängig, da die Boundingbox unberührt davon immer ein Rechteck ist. Die Laufzeit bei einem 128-eckigen Polygon liegt hier bei 23 Millisekunden. Die Polygonsubtraktion benötigt hingegen ca. 25 Sekunden zur Berechnung.

Anzahl Ecken	Dauer in [ms]	
	Polygonsubtraktion	getestete Methode
4	7.982	153
8	8.198	168
16	9.618	260
32	15.960	388
64	23.843	642
128	40.618	1.170

Tabelle 3.3: *Testergebnisse für Rechtecke, die das Polygon schneiden, in Millisekunden*

Schneiden die Rechtecke hingegen das Polygon, steigen die Laufzeiten beider Tests deutlich an. Die Laufzeit der Polygonsubtraktion steigt bei 128-Ecken auf 40 Sekunden an (siehe Tabelle 3.3). Diese Berechnung ist aufwendiger, da gemeinsame Schnittpunkte der beiden Formen gefunden werden müssen. Erst nach der Schnittpunktberechnung kann das subtrahierte Polygon erstellt werden. Auch die Laufzeit der Intersect-Methode steigt deutlich an. Die getestete Laufzeit beträgt 1,1 Sekunden. Nach der positiven Prüfung der Boundingbox müssen zudem die Intersect- und Contains-Methoden ausgeführt werden. Durch das Ausführen dieser beiden Methoden steigt die Laufzeit an. Innerhalb dieser wird genau bestimmt, ob die beiden Formen sich nur schneiden, oder ob das Rechteck komplett innerhalb des Polygons liegt.

Anzahl Ecken	Dauer in [ms]	
	Polygonsubtraktion	getestete Methode
4	3.882	559
8	4.718	827
16	6.210	1.418
32	9.563	2.656
64	14.585	5.111
128	25.155	10.007

Tabelle 3.4: *Testergebnisse für Rechtecke, die innerhalb des Polygons liegen, in Millisekunden*

Die Berechnung der Rechtecke, die vollständig im Polygon liegen, ist bei der zu testenden Methode die aufwendigste Testmenge. Tabelle 3.4 kann entnommen werden, dass der Algorithmus 10,0 Sekunden für die Prüfung der Rechtecke benötigt. Die Polygonsubtraktion hingegen erzielt ebenso schnell ein Ergebnis, wie bei den außenliegenden Polygonen (siehe Tabelle 3.2). Auch hier müssen bei der Subtraktion keine Schnittpunkte gefunden werden, um das Ergebnis-Polygon zu erstellen. Ein Polygon wird innerhalb einer Area über einen Pfad – bestehend aus seinen Eckkoordinaten – gespeichert. Dieser Pfad kann hier einfach um die neuen Punkte erweitert werden. Bei der neuen Methode hingegen verschuldet die Contains-Methode die gestiegene Laufzeit. Es reicht nicht aus innerhalb der Contains-Methode zu prüfen, ob jede Ecke des Rechtecks innerhalb des Polygons liegt. Dies wurde in Abbildung 3.1(a) verdeutlicht. Die Contains-Methode prüft zusätzlich, ob die Seiten des Rechteckes Schnittpunkte mit den Seiten des Polygons haben. Diese Berechnung muss daher mit jeder Seite des Polygons und mit jeder Seite des Rechteckes durchgeführt werden. Im aufwendigsten Fall - dass das Rechteck im Polygon liegt - muss folglich jede der vier Seiten des Rechteckes mit den 128 Seiten des Polygons geprüft werden. Trotz dieser Verzögerung ist die Laufzeit dieser Methode deutlich geringer als die Laufzeit der Polygonsubtraktion (25,1 Sekunden). Zusätzlich konnten bei keinem Test Differenzen beim Berechnungsergebnis gefunden werden. Die erwarteten Ergebnisse wurden vor den Testläufen dokumentiert. Die tatsächlichen Testergebnisse werden anschließend mit den erwarteten Ergebnissen verglichen. Hier traten ebenfalls keine Differenzen auf. Die getestete Methode berechnet folglich identische und korrekte Ergebnisse bei deutlich geringerer Laufzeit.

Gewichtung der Ergebnisse

In Abbildung 3.2 werden die Anteile von Octreeelementen des durch die DFS betreuten Luftraumes – gruppiert an der Anzahl der Sektoren – dargestellt. Dieser Octree besteht aus 3,1 Millionen Elementen und erstreckt sich über 1.086 Polygone [Ger16, S. 5].

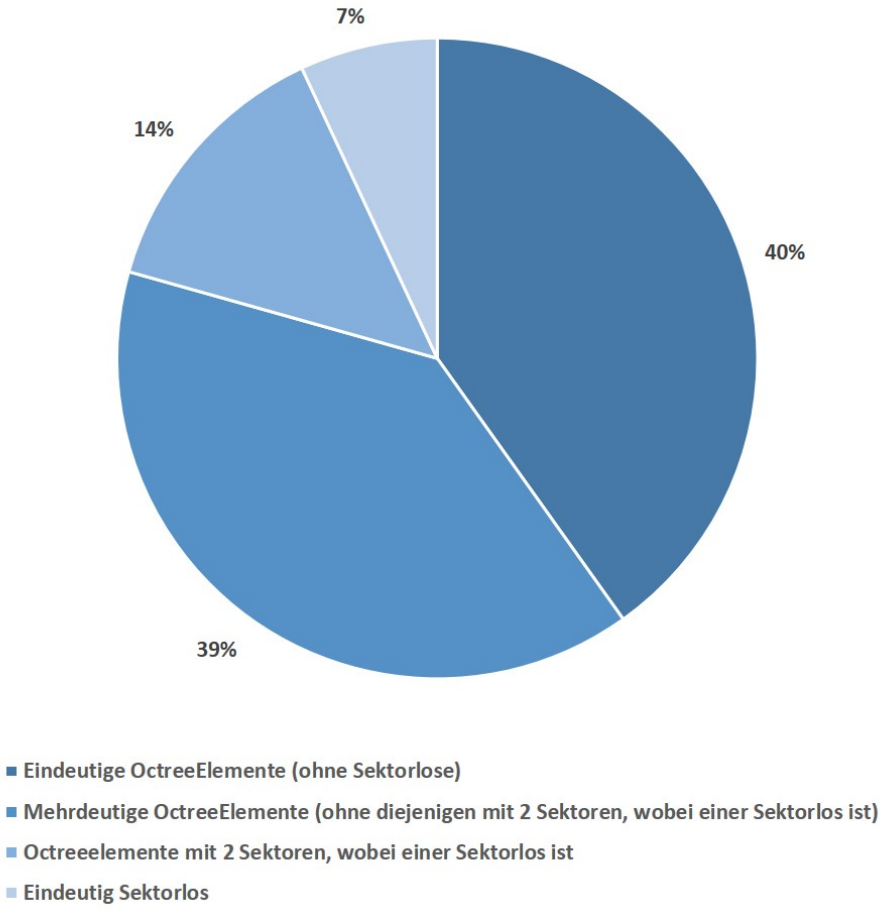


Abbildung 3.2: *Betrachtung der Verteilung der Eigenschaften von Elementen im Octree des DFS-Luftraumes hinsichtlich ihrer Eindeutigkeit*

40 % der Elemente des Baumes sind eindeutig einem Polygon (Sektor) zugeordnet. Im Umkehrschluss bedeutet das, dass 1.085 Polygone ausgeschlossen werden mussten. Und hier hat das beschriebene neue Verfahren entsprechend der Laufzeittests (siehe Tabelle 3.2) die kleinste Laufzeit. 39 % der Octreeelemente sind nicht eindeutig. Eine Prüfung, ob ein Sektor das Element komplett füllt ist nicht notwendig, da nicht nur ein Sektor in diesem Element liegt. Das Ausführen der Contains-Methode ist nicht notwendig. Dies verkürzt die Laufzeit der Erstellung der Elemente. Es gibt Octreeelemente, die nur einem Sektor zugewiesen werden können, welche jedoch nicht vollständig durch diesen gefüllt sind. Hierzu gibt es einen Platzhalter-Sektor mit dem Namen „Sektorlos“. Bei 14 % der Octreeelemente ist ein Platzhalter-Sektor notwendig.

Auch hier wurden mit der schnelleren Intersect-Methode alle Polygone ausgeschlossen. Erst dann wird – analog zu den eindeutigen – die Contains-Methode ausgeführt. Diese wird allerdings nicht vollständig ausgeführt, da das Polygon das Rechteck nicht füllt. Die verbleibenden 7 % der Elemente des Octrees konnten keinem Sektor zugeordnet werden. Hierbei wurden alle Sektoren über die Intersect-Methode ausgeschlossen. Eine Contains-Methode ist nicht notwendig, da es keinen verbleibenden Sektor gibt, der diese Octreeelemente schneidet.
Zusammenfassend muss die Contains-Methode bei ca. 1,7 Millionen Elementen vollständig ausgeführt werden. Das entspricht in etwa der Hälfte der gesamten Elemente. Im Vergleich dazu muss bei der vorherigen Methode jeder einzelne Sektor mit der Polygonsubtraktion verarbeitet und ausgeschlossen werden. Zudem müssen bei der vorherigen Funktion alle Octreeelemente – unabhängig der Eindeutigkeit – mit allen Sektoren geschnitten werden. Diese Herangehensweise verlängert den Aufbau des Octrees.

Quelldaten	Dauer in [ms]	
	bisherige Erzeugung	neue Erzeugung
STANLY	41.374	7.106
FRODO	64.029	9.094

Tabelle 3.5: *Dauer der Octree-Erstellung in Millisekunden*

Hinsichtlich der Testergebnisse wird die Polygonsubtraktion durch die hier getestete Methode ersetzt. Die Laufzeiten der fertigen Implementierung sind in Tabelle 3.5 aufgeführt. Bei der Integration der neuen Methode verbessert sich die Laufzeit des Aufbaus eines STANLY-Octrees von 41,3 Sekunden auf 7,1 Sekunden. Bei FRODO-Daten verbessert sich die Erzeugung des Octrees von 64 Sekunden auf neun Sekunden. Trotz der Verkürzung sind die Octrees identisch zu den Vorherigen. Dies wird geprüft, indem mit beiden Bäumen die Radardaten eines kompletten Tages ausgewertet werden. Hier treten keine Differenzen zu den Vorherigen auf. Zudem wird die Analyse der Element-Zahlen (siehe Abbildung 3.2) erneut mit den neu erstellten Octrees durchgeführt. Auch hier sind die Ergebnisse identisch.

3.2 Octreefactory

Die Sektorstrukturen des deutschen Luftraumes ändern sich regelmäßig. Beispielsweise werden regelmäßig im AIRAC-Zyklus (alle 28 Tage) neue Informationen über Sektoren bekannt gegeben [ica17]. Hierbei können sich sowohl Sektoren, als auch Airblocks verändern. Änderungen dieser Struktur haben zur Folge, dass auch ein Octree neu erzeugt werden muss. Eine Octreefactory soll diese Prozesse verwalten. Die Factory speichert jeden erstellten Octree in einer Datenbank und importiert diesen, wenn er benötigt wird. Hierbei hat jeder Octree, analog zu den Sektorkarten, eine Gültigkeit. Diese definiert den Lebenszyklus eines Octrees. Für jeden Zyklus ist zu Beginn nur das Start-Datum bekannt. Das Ende wird durch den Start des nächsten gültigen Octrees eingeleitet. Die Factory muss dem Statistikmodul immer den aktuellen Octree zur Verfügung stellen. Die Radardaten, die für die Statistiken ausgewertet werden, sollten immer mit dem aktuell gültigen Octree ausgewertet werden. Der Octree wird in Form eines Strings exportiert. Dieser wird vor dem Speichern erzeugt. Dieses Verfahren ist bereits in der Arbeit [Ger16] beschrieben. Zunächst muss geprüft werden, welche Datenbank für die Datenspeicherung der Octrees verwendet werden soll. Hierbei ist vor allem die Laufzeit des Ladens, sowie die Möglichkeit des Verwaltens der Octrees wichtig. Dies soll stärker gewertet werden als die Dauer eines Inserts. Zunächst werden nur die Datenbanken Oracle MySQL und Oracle NoSQL betrachtet. Diese werden durch das Lage- und Informationszentrum bereits im Support unterstützt.

3.2.1 Vergleich der Datenspeicherung

Für die Datenspeicherung der Octrees einer Octreefactory werden die Datenbanken Oracle MySQL und Oracle NoSQL betrachtet. Es gilt zu prüfen, welche der beiden Datenbanken hinsichtlich der Laufzeit und der Struktur besser für die Speicherung mehrerer Octree-Exporte geeignet ist.

Möglichkeiten der Datenspeicherung

Die Daten, die in der Datenbank gespeichert werden sollen, liegen als reiner Datenstring vor. Dieser Datenstring muss in der Tabelle nicht lesbar, sondern nur durch eine Octreefactory importierbar sein. Die Datenstrings sind zwischen 70 und 120 MB groß. Die Datenmenge hängt von der Herkunft der Sektorliste ab. Hierbei gibt es Daten aus den FRODO- (120 MB) und STANLY-Systemen (70 MB). Der Inhalt des Datenstrings ist in mehrere Abschnitte untergliedert. Eine Unterteilung oder einzelne Speicherung der Abschnitte innerhalb der Datenbank ist nicht notwendig. Die Abschnitte werden in der implementierten Import-Funktion als gesamter String verarbeitet. Da ein Octree 700 MB Arbeitsspeicher benötigt, soll nur der aktuelle Baum aus der Datenbank in die Factory geladen werden. Dies muss in der Abfrage der Octrees aus der Datenbank implementierbar sein. Der Fokus bei der Laufzeit-Betrachtung muss auf dem Lesen aus der Datenbank liegen. Hier soll der Octree bei einem Neustart schnell wiederhergestellt werden können. Das Speichern eines Octrees wird geringer gewichtet als das Lesen. Das Schreiben kann in einen gesonderten Thread ausgelagert werden und verzögert so die aktuelle Auswertung nicht.

Oracle NoSQL ist eine nicht-relationale Datenbank. Nicht-relationale Datenbanken speichern die Daten nicht zwangsläufig in Tabellen. Es gibt zwei für die Octree-Speicherung relevante nicht-relationale Datenbankschemata. Zum einen gibt es die dokumentenbasierte Speicherung. Hier werden Dokumente mit einem einheitlichen Format (bspw. JSON oder XML) abgelegt. Ein Dokument entspricht hierbei einem Datensatz. Die Informationen des Datensatzes – auch die eindeutige ID – sind innerhalb des gespeicherten Dokumentes festgehalten. Identifiziert werden die Datensätze durch die Informationen innerhalb des Dokumentes. Eine weitere Form sind Key-Value-Datenbanken. Hier werden die gespeicherten Daten einem festen Schlüssel zugeordnet. Einem Schlüssel wird genau ein Wert zugewiesen. Dieser Schlüssel ist vergleichbar mit dem Primary-Key einer relationalen Datenbank. Ein Datensatz mit mehreren Reihen muss in mehrere Subschlüssel aufgespalten werden. Es gibt keine fest definierte Filtersprache für die gezielte Abfrage von Datensätzen.

Relationale Datenbanken besitzen im Gegensatz dazu einen tabellarischen Aufbau. Datensätze werden in Zeilen in einer Tabelle gespeichert. Sie bieten die Möglichkeit Datensätze verschiedener Tabellen in eine Beziehung zu setzten. Jeder Datensatz verfügt über mehrere Datenfelder,

nach denen mittels SQL gefiltert werden kann. Diese Abfrage bietet die Möglichkeit gezielt Datensätze abzufragen, ohne dass die eindeutige ID bekannt ist. Oracle MySQL ist ein Beispiel für relationale Datenbanken [NSM+12, Had13].

Oracle NoSQL bietet unter anderem eine nicht-relationale Key-Value Datenbank [Ora17b]. Hier wird ausschließlich die Key-Value Funktionalität betrachet. Da diese Datenbank keine automatisch inkrementellen IDs erstellt, müssen diese innerhalb der Factory erzeugt werden. Dazu wird der Datenstring des Octree-Exports gehasht und als ID verwendet. Dies beugt beispielsweise dem doppelten Einfügen eines Baumes vor. Innerhalb der NoSQL-Datenbank werden die Daten diesen eindeutigen Schlüsseln zugewiesen. Der Schlüssel eines Octrees muss die eindeutige ID des Baumes beinhalten. Die Datenspeicherung und Schlüsselvergabe ist in Codebeispiel 3.2 dargestellt. Der Schlüssel einer NoSQL Datenbank kann aus einem Haupt- und einem Subschlüssel bestehen. Hierbei beschreibt der Hauptschlüssel beispielsweise eine Kategorie. Der Subschlüssel wird einem Hauptschlüssel zugewiesen. Er kann nähere Informationen zu dem gespeicherten Octree enthalten. Ein STANLY-Octree wird unter dem Hauptschlüssel „/Octree/STANLY/“ gespeichert. Unterhalb dieses Schlüssels werden alle Octrees in Subschlüsseln gespeichert. Diese Subschlüssel bestehen aus dem Gültigkeitsdatum und der eindeutigen ID des Octrees. Jeder zu speichernde Octree wird einem eindeutigen Schlüssel zugewiesen. Dieser Schlüssel setzt sich aus dem Haupt- und dem Subschlüssel zusammen. Hier muss zum Prüfen der Gültigkeit des Baumes nicht erst der Octree-Export geladen werden. Eine Abfrage-Semantik ist nicht vorhanden und muss so innerhalb der Factory implementiert werden. Zunächst wird dort eine Set mit allen Schlüsseln, die mit „Octree“ beginnen, erzeugt. Die Ergebnis-Set kann durchlaufen werden, bis das Datum – der erste Teil des Subschlüssels – größer als das aktuelle Datum ist. Da die Schlüssel innerhalb der Sets sortiert sind, ist der aktuell gültige Schlüssel der vorherige. Mit diesem gültigen Schlüssel wird anschließend der Octree aus der Datenbank geladen. Ab einer Größe von 1 MB empfiehlt sich die Speicherung eines Datensatzes als Large-Objekt (kurz LOB) in der Datenbank [Ora14].

```
1  Hauptschlüssel  – Subschlüssel
2  /Octree/Herkunft/– /datum          /id          /  –> Wert
3
4  /Octree/STANLY/ – /20170302010000/h0w4e54s.lob/ –> Octree–String
```

Quellcode 3.2: *NoSQL Schlüssel-Schema einer Octree-Datenbank*

Im Vergleich dazu besteht die MySQL-Datenbank aus einer Tabelle. Jeder Datensatz hat eine inkrementelle eindeutige Integer-ID und einen Timestamp ab dem der Octree gültig ist. Zudem enthält sie als Longblob den Datenstring des Octrees. Sollen bei einem Anwendungsstart die Octrees in eine Factory geladen werden, ist eine Abfragelogik notwendig. Pro Jahr werden mindestens zwölf Octrees gespeichert. Mittels SQL Semantik kann bei MySQL das Abfrageergebnis auf nur den aktuell gültigen Octree begrenzt werden. Dies kann über die Abfrage in Code 3.3 realisiert werden. Zusätzlich dazu ist es möglich über eine Abfrage zu kontrollieren, ob ein aktuellerer Octree in der Datenbank verfügbar ist.

```
1  SELECT *
2      FROM OCTREE
3      WHERE g_time <= now() & /*Gültigkeit ist erreicht*/
4          id != Factory.getAktId() & /*Nicht der aktuell verwendete Octree*/
5          g_time >= Factory.getAktGTime() /*Neuer als der aktuelle Octree*/
6      ORDER BY g_time DESC
7      LIMIT 1;
```

Quellcode 3.3: *SQL-Abfrage des aktuellen Octrees*

Vergleich der Möglichkeiten

Um die Laufzeiten beider Lösungen zu vergleichen, wurden Laufzeittests durchgeführt. Hierbei wurden sowohl STANLY-Daten, als auch die größeren FRODO-Daten verwendet. Die Testergebnisse sind in Tabelle 3.6 festgehalten. Das Speichern eines Octrees zeigt in der NoSQL-Datenbank eine schnellere Laufzeit. Diese beinhaltet das Hashen des Strings zur Erzeugung

eines eindeutigen Schlüssels. Im Schnitt benötigt das Speichern eines STANLY-Octrees hier nur 1,3 Sekunden. MySQL liegt mit einer Laufzeit von 1,75 Sekunden nur knapp dahinter. Beim Speichern eines FRODO-Octrees fällt der Unterschied größer aus. NoSQL kann die Daten in 5,4 Sekunden verarbeiten, wohingegen MySQL 7,7 Sekunden für die Speicherung in der Datenbank benötigt.

	Dauer in [ms]		
	Insert eines Octrees	Import eines Octrees	Import zweier Octrees
MySQL (STANLY)	1.752	3.288	6.020
NoSQL (STANLY)	1.306	5.054	9.887
MySQL (FRODO)	7.745	4.714	10.790
NoSQL (FRODO)	5.415	7.509	16.843

Tabelle 3.6: *Laufzeitvergleich in Millisekunden von Oracle MySQL und Oracle NoSQL beim Speichern und Einlesen eines Octrees (inkl. der Erzeugung des Baumes)*[5]

Bei der Betrachtung des Imports aus einer Datenbank fällt jedoch auf, dass MySQL beim Laden der Datenbank deutlich schneller als NoSQL agiert. Hierbei benötigt das Erstellen eines STANLY-Octrees aus einer MySQL-Datenbank nur 3,2 Sekunden. Aus einer NoSQL-Datenbank hingegen können STANLY-Octrees in fünf Sekunden in die Anwendung importiert werden. Das Laden von jeweils zwei STANLY-Octrees statt einem hat eine Laufzeiterhöhung auf sechs Sekunden (MySQL) und 9,89 Sekunden (NoSQL) zur Folge.
Beim Vergleich der Laufzeiten ist das Importieren und Lesen eines Octrees aus einer Datenbank stärker zu gewichten als das Speichern. Da Sektorkarten bereits vor dem Verwenden im System mindestens einen Monat vorher bekannt sind, kann der Octree deutlich vor der Verwendung erzeugt werden. Hierbei spielt die Laufzeit kaum eine Rolle. Das Laden hingegen passiert zur Laufzeit. Liegt z. B. nach einem Neustart kein Octree vor, müssen die eintreffenden Daten gehalten werden, bis dieser eingelesen ist. MySQL bietet darüber hinaus eine bessere Möglichkeit nach

[5]Die Laufzeitmessungen sind auf dem bereits genannten System durchgeführt worden. Jede Datenbank wurde in einem separaten Docker-Container ausgeführt.

Datensätzen zu filtern. Durch die tabellarische Auflistung und einer TIMESTAMP-Datenreihe können die Octrees einfacher verwaltet werden, als in einer Oracle NoSQL-Datenbank. Die Ergebnisse dieser Untersuchung spiegeln die Ergebnisse aus der Arbeit [NSM+12] wieder. Auch hier fällt das Ergebnis so aus, dass nicht-relationale Datenbanken für große Datenmengen geeignet sind, die häufiger geschrieben als gelesen werden. Im beschriebenen Anwendungsfall werden geringfügig (bis zu drei Bäume pro Monat) Daten in die Datenbank geschrieben. Jedoch muss regelmäßig geprüft werden, ob ein neuer Baum vorliegt. Dies kann durch die Tabellenstruktur und der möglichen Filterung in SQL-Statements der MySQL Datenbank schneller erfolgen.
Darüber hinaus kann auch eine NoSQL Lösung wie MongoDB verwendet werden. Hier hat besonders bei großen Datenmengen MongoDB einen Vorteil gegenüber MySQL [BRA12]. Um MongoDB für die Speicherung zu verwenden, müssen jedoch die Vorteile genau bestimmt werden. Diese Vorteile müssen anschließend mit dem steigenden Wartungsaufwand durch eine weitere Datenbank abgewogen werden. Dies muss in der weiteren Bearbeitung des Tools im Anschluss an diese Arbeit betrachtet werden.

Komprimieren des Exports

Im Anhang dieser Arbeit befinden sich Ausschnitte eines exemplarischen Exports von STANLY (Anhang A.1.1) und FRODO (Anhang A.1.2). Diese Dateien enthalten viele wiederholende Zeichenketten. Informationen, die einen solchen Aufbau und häufige nahe Wortwiederholungen aufweisen, sind für eine Kompression gut geeignet. Hier werden Redundanzen beispielsweise über eine Lookup Tabelle eliminiert. Java unterstützt nativ den Deflater-Algorithmus. Er basiert auf dem verlustfreien Gleitfenster-Algorithmus LZ77 [ML06, Sal07]. Dieser Algorithmus ist besonders für sich wiederholende Daten geeignet. Es handelt sich zudem um ein verlustfreies Kompressionsverfahren. Ein FRODO-String wird durch das Verfahren von ca. 130 MB auf 10 MB verkleinert. Bei einem STANLY-String verringert sich die Größe von ca. 70 MB auf 9 MB. Die Kompression verringert die Laufzeit der SQL-Statments sowohl bei einer Abfrage, als auch beim Einfügen.

	Dauer in [ms]		
	Insert eines Octrees	Import eines Octrees	Import zweier Octrees
STANLY	1.752	3.288	6.020
STANLY Compressed	1.721	2.343	5.420
FRODO	7.745	4.714	10.790
FRODO Compressed	2.538	3.765	6.455

Tabelle 3.7: *Laufzeitvergleich in Millisekunden von In- und Export von FRODO- und STANLY-Octrees, je unkomprimiert und komprimiert in MySQL (inkl. der Erzeugung des Baumes)*

In der Tabelle 3.7 sind die Laufzeitvorteile der Kompression dargestellt. Die Laufzeiten der unkomprimierten Übertragung wurden aus der vorherigen Tabelle 3.6 übernommen. Das Einfügen eines komprimierten STANLY-Octrees in die MySQL Datenbank ist nicht schneller, als das komprimierte Übertragen. Beide Methoden speichern den Octree in knapp über 1,7 Sekunden. Das Speichern eines FRODO-Octrees hingegen ist deutlich schneller. Hier hat sich die Laufzeit von 7,7 Sekunden auf 2,5 Sekunden minimiert. Beim Laden eines STANLY-Octrees verringert sich die Laufzeit von 3,2 Sekunden auf 2,3 Sekunden. Und auch bei einem FRODO-Sektor sinkt die Laufzeit von 4,7 Sekunden auf 3,7 Sekunden. Die Komprimierung wird aufgrund der Testergebnisse innerhalb des Datenbanktreibers verwendet, um Octrees in der MySQL Datenbank zu speichern. Zusätzlich spricht die reduzierte Datenmenge eines Octrees für die Komprimierung vor der Speicherung. Durch die Komprimierung kann anstatt eines Large-Blobs (maximal 2^{31} Bytes) ein Medium-Blob (maximal 2^{24} Bytes) für das Datenfeld verwendet werden [Ora17a].

3.2.2 Implementierung

Basierend auf den Testergebnissen aus Kapitel 3.2.1 wird die Octree-Factory mit einer MySQL-Datenbank verbunden. Für die Optimierung der Erzeugung eines Octrees aus der Datenbank wurde der Import verändert. Dieser erhält als Übergabeparameter nicht wie ursprünglich einen String, sondern einen InputStream. Dieser wird von beiden Datenbanken (sowohl NoSQL

als auch MySQL) beim Lesen eines „Medium Blob“ Attributes zurückgegeben. Der erhaltene InputStream muss so nicht in einen String umgewandelt werden.
Zusätzlich wird ein Datentyp Octree implementiert. Dieser speichert jeweils den Root-Knoten eines erzeugten Octrees. Außerdem erhält jeder Baum eine eindeutige Integer ID, die den Baum in der Datenbank identifiziert. Darüber hinaus hat er den Start seiner Gültigkeit als java.util.Date gespeichert. Die OctreeFactory speichert je einen Octree für STANLY- und einen für FRODO-Sektoren. Zusätzlich dazu gibt es einen DataBaseConnector, über den die Factory mit der MySQL-Datenbank Daten austauscht.
Beim Start der Factory lädt sie die beiden aktuellen Octrees – FRODO und STANLY – aus der Datenbank. Sie bietet über die Methode addOctElement(Octree) zudem die Möglichkeit einen Octree hinzuzufügen. Dieser wird nicht innerhalb der Factory selbst gespeichert. Er wird in einem separaten Thread direkt in die Datenbank geschrieben. Hier bietet die Methode insert(Octree) des DatabaseConnectors die entsprechende Schnittstellen-Methode. Die Factory selbst lädt zu jedem Zeitpunkt nur den jeweils aktuellen Octree für STANLY- und FRODO-Auswertungen. Diese werden zunächst in zwei zusätzliche Octree-Variablen gespeichert. Sobald die geladenen Bäume ihre Gültigkeit erreicht haben, werden die Adressen der Octrees mit denen der bis dato aktuellen Octrees getauscht. Um stets den aktuellen Octree gespeichert zu haben, soll zu jedem neuen Statistikslot (zunächst jede Stunde) geprüft werden, ob ein neuer Octree vorhanden ist. Hierbei übernimmt das Statistikmodul die Aufgabe, die Octreefactory über den neuen Slot zu informieren. Ist ein neuer Octree verfügbar, wird dieser innerhalb der OctreeFactory über die Methode getAktOctrees() in einem gesonderten Thread geladen. Die ungültig gewordenen Octrees werden dann, durch den Aufruf enterNextSlot() ersetzt.
Im Live-Betrieb wird kurz vor einem Slot-Ende in der Octreefactory geprüft, ob ein neuer Octree gültig ist. Dies soll jedoch nicht bei jedem Aufruf der Factory passieren. Durch das Statistikmodul wird 60 Sekunden vor dem Ablaufen eines Statistikslots (jede volle Stunde) die Methode Octreefactory.renew() aufgerufen. Diese prüft in einem gesonderten Thread, ob ein neuer Octree für den nächsten Slot vorliegt. Gibt es einen neuen Octree, wird dieser aus der Datenbank geladen und wiederhergestellt. Ist die Gültigkeit des neuen Octrees erreicht, dann muss lediglich die Adresse der Variable des aktuellen Octrees in die des Neuen geändert werden. Dass der neue Slot erreicht ist, wird der Factory über die Methode switchSlot() mitgeteilt. Hier

werden lediglich die Variablen vertauscht. Ausgeführt wird sie vom Statistikmodul, wenn dieses einen Slotwechsel feststellt.
Die Renew Methode greift über den DatabaseConnector auf die MySQL Datenbank zu. Hier benutzt sie die Methode DatabaseConnector.getCurrOctree(). Innerhalb dieser Methode wird die bereits in Quellcode 3.3 dargestellte SQL-Abfrage verwendet.
Die Komprimierung wird im DatabaseConnector umgesetzt. Hier wird der String komprimiert bevor er in die Datenbank geschrieben wird. Für das Verwenden des Connectors gibt es folglich keine Veränderungen. Die Daten werden aus MySQL über einen InputStream zurückgegeben. Dieser InputStream wird durch einen DeflaterInputStream ersetzt und dann vom Connector zurückgegeben. So kann die Factory den InputStream wie bisher verwenden, er wird allerdings erst beim Lesen dekomprimiert.

3.3 Fazit

Die Laufzeit der Erstellung eines Octrees für STANLY-Sektoren wurde von 40 Sekunden auf 7,1 Sekunden verbessert [Ger16]. Diese Verbesserung der Laufzeit um 82 % basiert auf der Ablösung der bisher verwendeten Polygonsubtraktion. Hier wird nun ein Verfahren verwendet, welches eine schnelle Vorauswahl durchführt. Nach der Vorauswahl werden effizientere Methoden als die bisherige Polygonsubtraktion verwendet um einen Schnitt zu detektieren.
Zudem wird eine Octreefactory entwickelt, welche verschiedene Octree-Bäume verwaltet und an laufende Anwendungen verteilt. Hierbei werden die gespeicherten Bäume in einer MySQL-Datenbank gespeichert. Die gespeicherten Export-Daten werden vor dem Speichern über das Deflater-Verfahren komprimiert. Das führt zu einer Reduktion der Datenmenge eines STANLY-Octree-Exports von 70 MB auf 9 MB. Dies führt zu einer Verringerung der Importzeit eines STANLY-Octrees aus einer MySQL-Datenbank von 3,2 Sekunden auf 2,3 Sekunden, was einer Verbesserung von 28 % entspricht.

4 Erweitern um ein Statistikmodul

Im folgenden Kapitel wird auf das Statistikmodul eingegangen. Hierbei werden zunächst die Anforderungen genannt. Anschließend wird die Implementierung beleuchtet. Abschließend wird das entwickelte Modul getestet, indem es mit STANLY verglichen wird.

4.1 Analyse der Anforderungen

Die Statistiken, die erhoben werden sollen, bestehen aus zwei Bestandteilen. Diese sind zum einen die Auslastungsstatistiken und zum anderen die eventbasierten Statistiken. Beide werden in den folgenden Abschnitten näher erläutert. Zudem muss das Statistikmodul Sektorkonfigurationen betrachten. Auch dies wird in diesem Kapitel beschrieben.

4.1.1 Auslastungsstatistiken

Eine Auslastungsstatistik soll die Auslastung eines Sektors durch Luftfahrzeuge abbilden. Hierbei muss die Statistik stets einen Zeitraum t bis t+Δt betrachten. Ein Tag wird in mehre-

re Zeiträume unterteilt. Diese werden als Slot bezeichnet. Die Slotdauer soll individuell im Statistikmodul konfiguriert werden. Getestet und entwickelt wird sie mit einer Dauer von einer Stunde, da das Altsystem ebenfalls diese Slotdauer verwendet. Für jeden Slot wird aufgezeichnet, wie viele Luftfahrzeuge in den jeweiligen Sektor ein- und ausgetreten sind. Zusätzlich hierzu soll betrachtet werden, wie viele verschiedene Luftfahrzeuge innerhalb des jeweiligen Sektors geflogen sind. Ein Beispiel für eine solche Auslastungsstatistik ist in der folgenden Tabelle aufgeführt:

slotStartTime	slotEndTime	numOfEinfluege	numOfAusfluege	numOfAircraft
00:00:00.000	00:59:59.999	5	3	5
01:00:00.000	01:59:59.999	0	2	2
[...]	[...]	[...]	[...]	[...]
12:00:00.000	12:59:59.999	33	35	40
13:00:00.000	13:59:59.999	26	25	31
[...]	[...]	[...]	[...]	[...]
22:00:00.000	22:59:59.999	11	12	13
23:00:00.000	23:59:59.999	4	5	5

Tabelle 4.1: *Beispielhafte tabellarische Darstellung einer Auslastungsstatistik für den Sektor EDUUERL22*

4.1.2 Eventbasierte Statistiken

Eventbasierte Statistiken bilden Flugereignisse ab. Als Flugereignisse zählen hierbei beispielsweise Eintritte in und Austritte aus einem Sektor. Zur Laufzeit müssen alle Radarpositionen einer Strecke jedes Fluges einem Sektor zugeordnet werden. Dabei muss auch jeder Sektorwechsel gespeichert werden. Eine spätere Erweiterung um die Dokumentation von Starts und Landungen ist ebenfalls angedacht. Hierzu ist eine Analyse der Flugspur notwendig. Eventbasierte Statistiken müssen folgende Informationen dokumentieren:

- Eindeutige Flugidentifikation
- Eventtyp (Eintritt, Austritt)
- Betroffener Sektor
- Indirekt betroffener Sektor
- Zeitpunkt des Events
- Position des Events
- Zurückgelegte Strecke seit dem letzten Event

4.1.3 Sektorkonfigurationen

Wie bereits in Kapitel 2.1 beschrieben wurde, können verkehrsschwache Sektoren gleichzeitig betrieben und zusammengelegt werden. Dies verändert die Betrachtung der Statistiken. Die Auslastungsstatistik einzelner Sektoren kann möglicherweise keine Aussage darüber treffen, wie ausgelastet ein Lotse wirklich ist. Sollten Sektoren zusammengelegt werden, ist es notwendig Auslastungsstatistiken für die jeweilige Sektorkonfiguration zu erheben. Nichtsdestotrotz werden die Statistiken einzelner Sektoren weiterhin benötigt, um die erzeugten Daten mit anderen Sektorkonfigurationen zu vergleichen.
Dies muss auch bei der Erstellung der eventbasierten Statistiken bedacht werden. Bei Nichtbeachtung der Sektorkonfigurationen sind in den Statistiken Sektorübertritte aufgeführt, die keinerlei Auswirkungen für den verantwortlichen Lotsen haben. Es gibt daher keine Übergabe des Flugzeuges an einen anderen Lotsen. Werden hingegen ausschließlich die Sektorkonfigurationen betrachtet, können die erstellten Statistikdaten nachträglich nicht mit einer anderen Sektorkonfiguration ausgewertet werden.
Entsprechend dieser Betrachtung müssen die Statistiken sowohl für die jeweiligen Sektorkonfigurationen, als auch für die einzelnen Sektoren erhoben werden. Dies führt zu einer Verbesserung der Datenqualität sowie einer besseren Möglichkeit zur weiteren Datenanalyse.

4.2 Implementierung des Statistikmoduls

In der Anwendung werden die eventbasierten Statistiken in Objekten des Typs Aufenthalt gespeichert. Diese speichern je ein Ein- und Austrittsevent. Zudem sind sie einem Flug und einem Sektor zugewiesen. Innerhalb des Aufenthalts wird die geflogene Strecke zwischen einer neuen und der letzten Koordinate auf die bisher geflogene Strecke addiert. Die enthaltenen Events speichern jeweils den betroffenen Flug, den Typ des Events, den betroffenen Sektor sowie – je nach Typ – den vorherigen oder neuen Sektor. Der Typ eines Events bezieht sich hierbei auf die Information, ob es sich um einen Ein- oder Austritt handelt. Darüber hinaus wird der Radarpunkt gespeichert, bei dem das Event ausgelöst wurde. Der Radarpunkt beinhaltet die eindeutige ID sowie das Callsign des Fluges. Zudem wird der Zeitpunkt in Form einer Date-Variable gespeichert. Zusätzlich wird die Flugposition gespeichert. Diese beinhaltet sowohl die Lat/Long-Koordinaten für einen späteren Export, als auch die projizierten Koordinaten für die Berechnungen. Die Lat/Long-Koordinaten werden über eine stereografische Projektion in kartesische Koordinaten umgerechnet [Sos99]. Einlaufende Koordinaten in das Statistikmodul werden als Radarpunkte übergeben. Sie beinhalten alle Informationen, die notwendig für die Berechnung sind. Das UML-Klassen-Schema der Anwendung ist in Abbildung 4.1 vereinfacht dargestellt. Darüber hinaus ist im Anhang unter A.2 das in [Ger16] beschriebene UML-Diagramm der Octree-Struktur zu finden.

Die Auslastungsstatistiken werden innerhalb der Sektoren gespeichert. Hier gibt es eine Sektor-Live Klasse, die einen Sektor um seine Statistiken erweitert. Dies entspricht dem Decorator Pattern [Hau10]. Diese werden in eine Liste von SektorStats abgelegt. SektorStats repräsentieren je einen Statistikslot für einen Sektor. Sie haben einen Start- und ein Endzeitpunkt, die jeweils als Date gespeichert werden. Sie haben zudem je eine Liste von Flügen für eingetretene und ausgetretene Flüge. Die geflogenen Flüge werden in einem Set gespeichert. So werden Flüge, die mehrfach einfliegen, nicht doppelt gezählt. Die Längen dieser Listen ergeben schließlich die Anzahl der Flüge für den betrachteten Slot.

Abbildung 4.1: *Vereinfachtes UML-Diagramm des Statistikmoduls*

Soll eine Statistik für eine einzelne Radardatei erstellt werden, muss zunächst ein fortlaufender Radarstrom erstellt werden. Hierbei werden aus der Quelldatei alle Radarpunkte gelesen und in ein Array gespeichert. Dieses Array wird anschließend durchlaufen. Die Radarkoordinaten werden sowohl innerhalb der Online- als auch der Offline-Version über die Methode sendRadarPunkt() an das Statistikmodul weiter gegeben. Dieses Vorgehen wird bei der Online-Version durch einen Apache Camel Empfänger ausgeführt. Dieser empfängt die von EDIS versendeten Koordinaten und gibt sie an das Statistikmodul weiter. Ein Objekt der Klasse Statistikverwalter repräsentiert eine Instanz des Statistikmoduls. Pro zu erstellender Statistik (Tages- oder Live-Statistiken) wird ein Statistikverwalter verwendet, um die Statistiken zu berechnen. Bei der Offline-Auswertung ist der Statistikveralter als Singleton implementiert.
Das Statistikmodul hat eine Map für die Speicherung aller aktuell bekannten und nicht abgeschlossenen Flüge. Hier werden die Flüge einem eindeutigen Long-Schlüssel zugeordnet. Dieser Schlüssel ist für jeden Flug eindeutig und wird mit den Radardaten übertragen. Empfängt das Statistikmodul eine Radarkoordinate, wird mit dem Flugschlüssel der Koordinate innerhalb der Map der entsprechende Flug gesucht. Ist innerhalb der Map kein Flug mit diesem Schlüssel vorhanden, wird ein Neuer erstellt und hinzugefügt. Die empfangenen Koordinaten werden dann dem Flug über die Methode newPosition() übergeben. Die Radarposition wird innerhalb des Fluges weiterverarbeitet.
Die Flug-Objekt-Instanz nimmt die neue Koordinate entgegen. Jede neue Koordinate wird durch den übergebenen Octree einem Sektor zugeordnet. Anschließend wird geprüft, ob der zugewiesene Sektor sich vom bisherigen unterscheidet, oder sogar der erste gefundene Sektor innerhalb des deutschen Luftraumes ist. Handelt es sich um den ersten Sektor eines Fluges, wird ein Aufenthalt-Objekt erstellt. Dieses hat zunächst nur das Eintrittsevent, das ebenfalls erstellt wird. Liegt allerdings ein Sektorwechsel vor, wird zunächst ein Austrittsevent erstellt und dem bisherigen Aufenthalt-Objekt übergeben. Anschließend wird das alte Aufenthalt-Objekt durch ein neues mit ebenfalls neuem Eintrittevent ersetzt. Bei jedem erkannten Sektorwechsel muss außerdem geprüft werden, ob sich die Sektorkonfiguration des neuen und des alten Sektors unterscheiden. Werden die beiden Sektoren in verschiedenen Sektorkonfigurationen betrieben, wird hier ein Sektorkonfigurationswechsel ausgelöst. Die Events der Sektorkonfigurationen werden analog zu den beschriebenen Events erstellt. Haben der bisherige und der neue Sektor

die identische Konfiguration, wird keine Änderung durchgeführt. Beim Sektorwechsel wird über die Funktion sektor.getSektorStats(aktuelleZeit) das aktuelle SektorStats-Objekt für den neuen und den alten Sektor geladen. Innerhalb dieser werden die Flüge in die Ein- und Austrittslisten eingetragen. Zudem wird ein Flug in das Set Fluege eingetragen.
In jedem Fall – unabhängig vom Sektor – werden die bisherige Koordinate und der Sektor des Flugobjektes anschließend durch die neuen Werte ersetzt. Zudem wird innerhalb des aktuellen Aufenthalts stets die Strecke zwischen der letzten und der neuen Koordinate zu der bisherigen Distanz addiert. Für jeden Flug muss auch ohne Sektorwechsel kontrolliert werden, ob die gespeicherten SektorStats noch gültig sind. Ändert sich der Slot, muss der Flug den neuen Slot speichern und sich in dessen Flugliste eintragen. Innerhalb der getSektorStats()-Methode wird zunächst geprüft, ob für den angefragten Slot bereits eine Statistik vorhanden ist. Liegt keine Statistik vor, wird eine Neue erzeugt. Hatte ein Sektor im Zeitraum eines Slots keinen Verkehr, wurde für den Sektor kein Slot erstellt. Diese entstehenden Lücken werden gefüllt: Wenn ein Sektor mehrere Stunden keinen Flugverkehr hatte, werden die Statistiken trotzdem im Nachhinein erstellt.
Innerhalb der verwendeten Testdaten für die Entwicklung reißen Spuren teilweise innerhalb des deutschen Luftraumes ab. Reißt die Spur eines Fluges innerhalb eines Sektors ab, würden so die Sektoraustritte nicht inkrementiert werden. Außerdem würde es für den Flug kein Austrittsevent aus dem Luftraum geben. Dies verfälscht die Statistik. Bei Landungen innerhalb des deutschen Luftraumes tritt dieselbe Problematik auf. Hierfür muss das Statistik-Berechnungs-Modul einen Automatismus haben, der veraltete Objekte erkennt. Für die Verwaltung dieser Flüge wird eine feste Timeout-Zeit von 60 Sekunden definiert. Wird ein Objekt gefunden, welches länger als 60 Sekunden keine neuen Koordinaten gesendet hat, wird der Flug abgeschlossen. Hierbei werden Sektoraustritte innerhalb des letzten Slots, in dem sich der Flug befunden hat, erhöht. Außerdem wird ein Austrittsevent für die letzte empfangene Koordinate erstellt. Dieser Automatismus wird durch das Statistikverwalter-Modul für alle bekannten Flüge ausgeführt. Innerhalb der Offline-Version wird er durch das Aufrufen der Methode checkFlights() zum Ende des übergebenen Radardatensatzes ausgeführt. Innerhalb der späteren Live-Version wird diese Methode periodisch ausgelöst.
Innerhalb der Offline-Version werden nach dem Erstellen der Statistiken alle gespeicherten

und erhobenen Daten in CSV-Dateien gespeichert. Hierbei wird die Auslastungsstatistik in je einer Datei pro Sektor und Tag ausgegeben. Die eventbasierten Statistiken hingegen werden gesammelt als Liste in eine CSV-Datei geschrieben. Diese beinhaltet alle Events des betrachteten Zeitraumes. Innerhalb der späteren Live-Version können erst Datensätze gespeichert werden, die bereits vollständig sind. Hierbei können z. B. Sektorstatistiken erst dann gespeichert werden, wenn sichergestellt ist, dass sie abgeschlossen sind. Gemäß der checkFlights()-Methode kann sich die Statistik auch nach Ende des eigentlichen Slots ändern. Die Sektoraufenthalte der eventbasierten Statistiken können erst dann gespeichert werden, wenn das Flugzeug aus dem entsprechenden Sektor ausgetreten ist. Der letzte Aufenthalt eines Fluges kann ebenfalls durch die checkFlights()-Methode nachträglich geändert werden. Werden alle Events eines abgeschlossen Fluges exportiert, können die gespeicherten Informationen freigegeben werden. Sie werden für eine weitere Berechnung nicht benötigt. Auch die einzelnen Slots und Tagesstatistiken können nach dem Speichern freigegeben werden. Für die Datenspeicherung der Statistiken kann eine Datenbank verwendet werden. Dies ist in Abschnitt 4.2.1 beschrieben. Der implementierte Export der Offline-Version benötigt 1,5 Sekunden.
Einige Teile der beschriebenen Implementierung müssen näher beleuchtet werden. Hierzu zählen die Implementierungen der Persistenz und der Streckenberechnung. Außerdem wird die Genauigkeit der eventbasierten Statistiken beleuchtet.

4.2.1 Persistenzimplementierung

Die erstellten Statistiken des Moduls müssen persistent abgelegt werden. Hierzu ist ein CSV-Export implementiert. Die Daten sollen hingegen in eine Datenbank geschrieben werden. Dies bietet im Bezug auf die Auswertung der Statistiken durch die Abfragesyntax einen großen Komfort. Zunächst sollen die erhobenen Daten in einer MySQL Datenbank gespeichert werden. Ein mögliches Datenbankschema ist in Abbildung 4.2 dargestellt. Alle dargestellten Verbindungen stehen für eine „1:n“ Beziehung. Am Beispiel der Verbindung zwischen den Tabellen Aufenthalt und Flight bedeutet dies, dass ein Flug mehrere Aufenthalte haben kann.

Ein Aufenthalt gehört jedoch zu exakt einem Flug. Es kann jedoch auch Flüge geben, die keinen Aufenthalt haben. Die durchgezogenen Verbindungen stehen zusätzlich dazu für eine notwendige Beziehung. Im gezeigten Beispiel ist diese Verbindung z. B. zwischen Sectorconfig und Configsektor. Jede Sektorkonfiguration muss mindestens einmal einem Sektor in der Tabelle Configsektor zugewiesen werden. Die Konfiguration kann jedoch auch aus mehreren Sektoren bestehen.

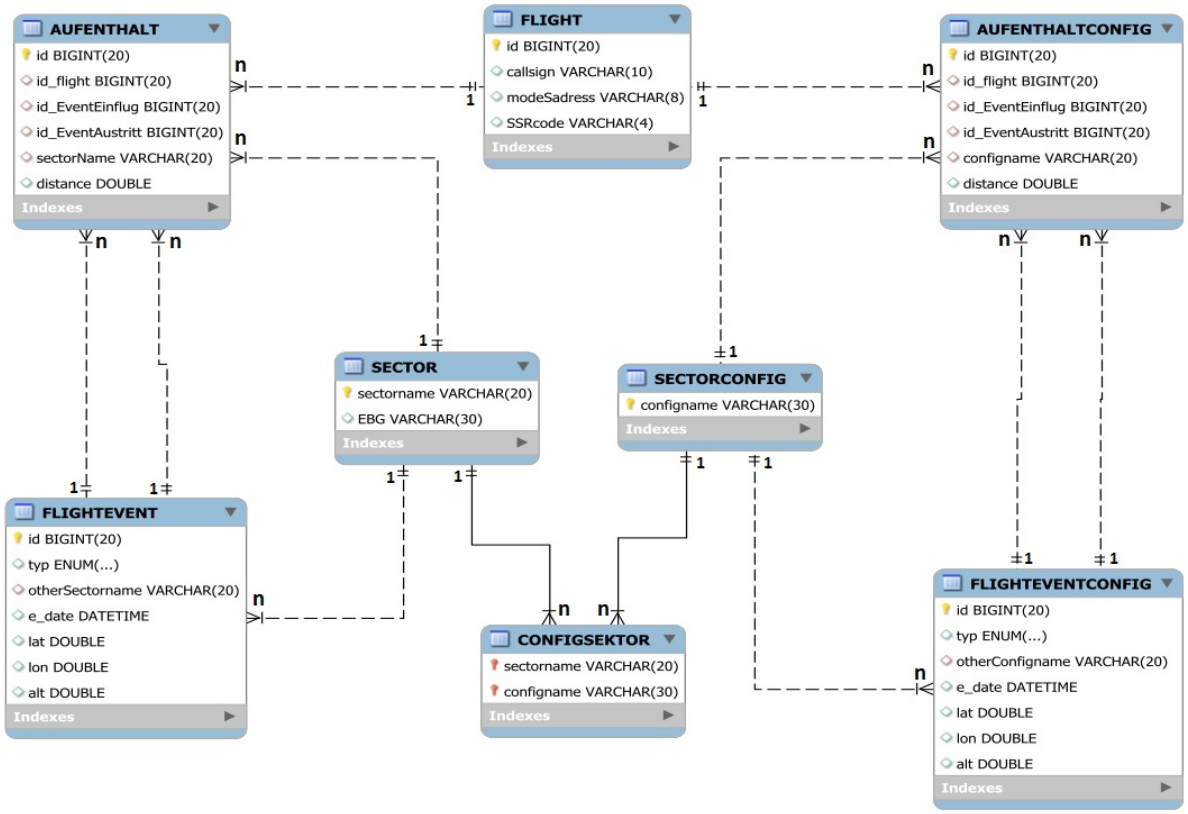


Abbildung 4.2: *Datenbankschema der MySQL-Datenbank zur Speicherung der erzeugten Flugevents*

Für die Radar-Koordinaten vom 22.05.2015 werden ca. 120.000 Flugevents erstellt und gespeichert. Das Speichern dieser Events innerhalb einer CSV-Datei benötigt 1,5 Sekunden. Dieser Dateiexport wurde testweise durch eine JDBC-Verbindung an eine Oracle MySQL Datenbank ersetzt. Analog zum Export werden hier alle Daten nach dem Erzeugen der Tagesstatistiken in die Datenbank geschrieben. Die Schreiboperationen benötigen 30,7 Sekunden (siehe Ta-

belle 4.2). Die 120.000 Flugevents werden hierbei in knapp 200.000 Datensätzen gespeichert. Darunter sind nicht nur die jeweiligen Flugevents sondern auch die jeweiligen Aufenthalte, Flüge und Sektoren. Die Ereignisse wurden hier nach dem Abschluss der Statistiken in die Datenbank gespeichert. Hier wurden alle Datensätze final hintereinander in die Datenbank geschrieben. Die Autocommit-Funktion wurde dafür im Databaseconnector deaktiviert.

Verfahren	Dauer in [ms]	
	Statistikerstellung	Speicherung
CSV - File	5.197	1.548
JDBC MySQL	5.231	30.699
Eclipse Link MySQL	5.961	52.825
OpenJPA MySQL	7.979	39.504

Tabelle 4.2: *Laufzeiten der Speicherung der eventbasierten Statistiken in Millisekunden*

Eine Alternative zu JDBC stellt die Java Persistence API (kurz JPA) dar. JPA stellt einheitliche Methoden zur Speicherung von Objekten zur Verfügung. Zu speichernde Objekte werden in je einer Tabelle pro Klasse mit definierten Spalten abgelegt. Die zu speichernden Objekte werden in einem EntityManager vermerkt. Innerhalb des Programmes können sogenannte EntityTransactions ausgeführt werden. Hier werden alle an den Objekten – die im EntityManager vermerkt sind – vorgenommenen Änderungen in der Datenbank persistiert. Objekte, die nicht weiter in der Datenbank verändert werden sollen – beispielsweise bei einem abgeschlossenen Flug – können wieder aus dem EntityManager entfernt werden. Eine gesonderte Speicherung über SQL-Statments, wie bei JDBC, ist nicht notwendig. Im Rahmen dieser Arbeit wurden die zwei JPA-Provider EclipseLink und Apache OpenJPA näher betrachtet.
Diese Provider werden innerhalb einer Arbeit [Šem12] von Šembera verglichen. Beim Laufzeitvergleich liegt der Fokus auf dem Einfügen von Datensätzen. Die geschriebenen Statistiken werden durch das Statistikmodul nicht wieder eingelesen. Šembera zeigt deutliche Laufzeitvorteile von EclipseLink gegenüber OpenJPA. EclipseLink stellt sich hier als schnellste Lösung für Insert-Operationen in eine MySQL-Datenbank heraus. OpenJPA benötigt laut den Tests fast dreimal länger. Auch Wegrzynowicz bestätigt in [Weg12] EclipseLink bei der Laufzeit als

bessere Lösung. Um die Laufzeiten für das Statistikmodul selbst zu bestimmen, wurde sowohl eine OpenJPA als auch eine EclipseLink Implementierung erarbeitet.
Es wurden die Tagesstatistiken für den 22.05.2015 erstellt und die Laufzeiten der Datenspeicherung gemessen. Hier zeigte sich – im Gegensatz zu den oben genannten Vergleichsarbeiten – eine kürzere Laufzeit bei OpenJPA. Die Ergebnisse sind bereits in Tabelle 4.2 aufgeführt. EclipseLink benötigt für die Speicherung der Tageskoordinaten 13 Sekunden länger als OpenJPA. OpenJPA hingegen verlängert die Statistikerstellung im Vergleich zu EclipseLink um zwei Sekunden. Beim Vergleich der gespeicherten Daten innerhalb der Datenbank liegen keine Unterschiede vor. OpenJPA benötigt neun Sekunden länger als eine Anbindung mit JDBC. OpenJPA bringt jedoch den Vorteil mit, dass die Verwaltung der Änderungen nicht durch den Entwickler erfolgen muss. Dieser muss ausschließlich festlegen, welche Objekte gespeichert werden müssen. Dies hat den Vorteil, dass Änderungen von Objekten, die bereits in der Datenbank gespeichert sind, nicht manuell über ein UPDATE-Statment erfolgen müssen. Die Änderungen werden automatisch durch OpenJPA erkannt und ausgeführt.
Resultierend aus dieser Betrachtung wird die Persistenz im Live-Betrieb mittels OpenJPA durchgeführt. Die Speicherung der Statistiken erfolgt nicht einmalig, sondern verteilt auf den Tag. Hier werden regelmäßig Transaktionen mit JPA ausgelöst. Abgeschlossene Flüge werden auf der Transaktionsliste entfernt.

4.2.2 Genauigkeit

Für jedes Luftfahrzeug, das von der Statistik erfasst werden soll, laufen in EDIS in einem Intervall von fünf Sekunden Radarpositionen ein. Auf Grundlage dieser Positionen werden die Statistiken erstellt. Ziel ist es den Ein- und Austrittpunkt eines Sektorwechsels genauer, als die fünf Sekunden Abdeckung, festzustellen.

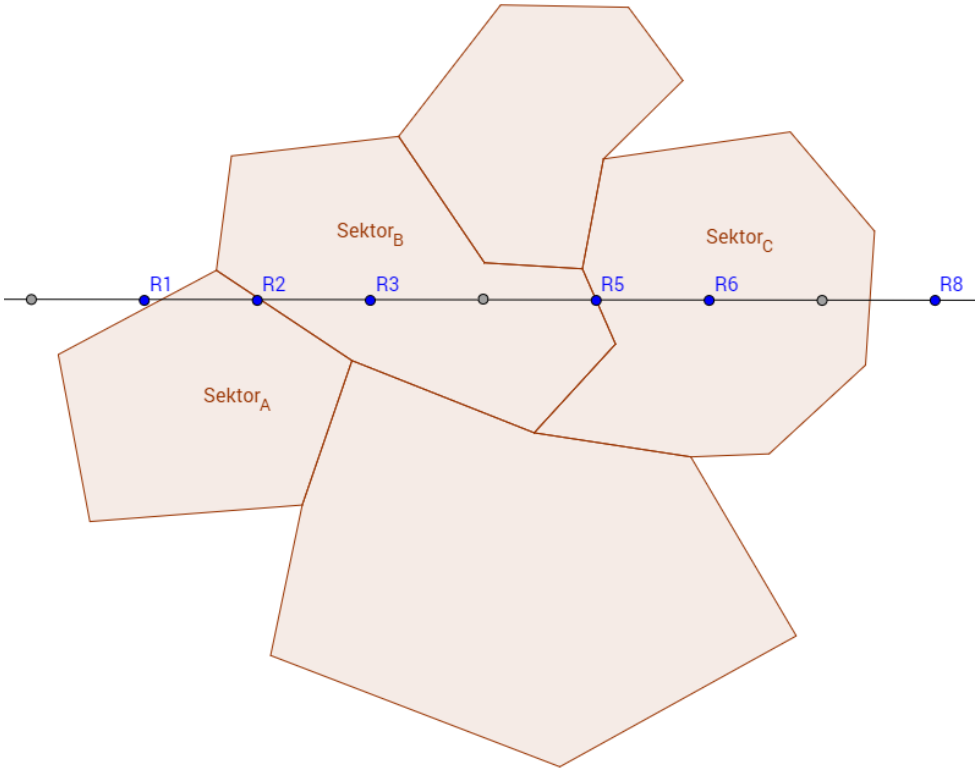


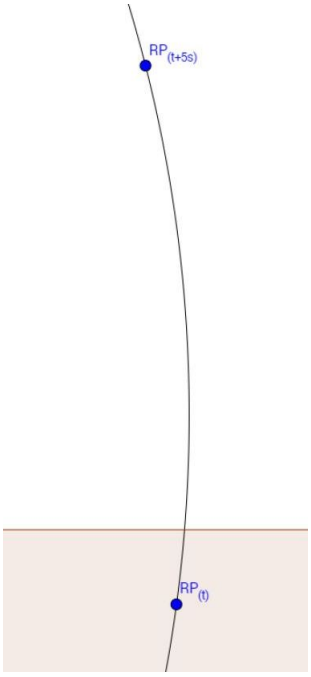
Abbildung 4.3: *Beispielhafte Radarspur eines Fluges durch hypothetische Sektoren*

Abbildung 4.3 zeigt eine Problematik der LIZ-Radarversorgung für Statistiken. Zu sehen sind in der Abbildung eine hypothetische Sektor-Struktur und ein Radarverlauf eines Fluges. Hierbei sind die betrachteten Radarpunkte blau markiert. Die Radarspur von F verläuft von links nach rechts.

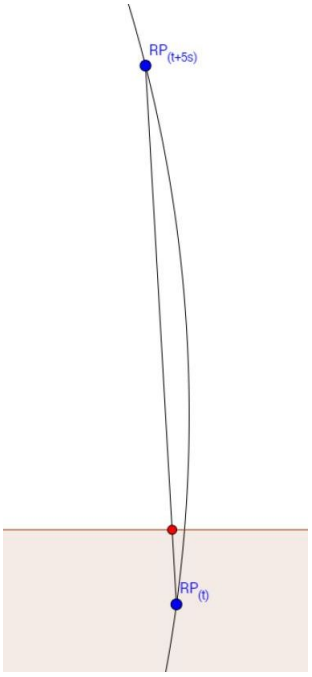
Der Flug tritt zuerst in den Sektor A ein. Die letzte Radarposition außerhalb der Struktur (R1) liegt nah an der Sektorgrenze zu A. Das hat zur Folge, dass der Sektor Eintritt erst bei Punkt R2 erkannt werden kann. Dieser liegt jedoch bereits nahe der Sektorgrenze des nächsten Sektors. Wird der Sektorwechsel bemerkt, muss festgelegt werden, an welcher Position der Flug den Sektor gewechselt hat. Der Sektor Ein- und Austrittspunkt ist diejenige Position, an der das Flugzeug zwischen zwei Sektoren über die gemeinsame Grenze hinausfliegt. Fliegt ein Flugzeug exakt auf der Sektorgrenze, findet noch kein Sektorwechsel statt. Hierbei ist daher die erste Radarposition R in einem neuen Sektor die Eintrittsposition in diesen. Diese Position ist schließlich auch die Austrittsposition des alten Sektors. Im abgebildeten Beispiel führt dies jedoch zu einer Ungenauigkeit. Hier kann unter anderem im Fall von R2 und R3 gezeigt werden,

dass die Ein- und Austritte um bis zu fünf Sekunden verschoben sind. Insbesondere bei der Berechnung der zurückgelegten Strecke führt dies zu einer Ungenauigkeit. Diese entspricht der Strecke, die ein Verkehrsflugzeug in fünf Sekunden fliegt. Bei einem Airbus A320 beträgt die Reisegeschwindigkeit 858 km/h [Luf17]. In fünf Sekunden entspricht das einer Strecke von ca. 1,2 km.

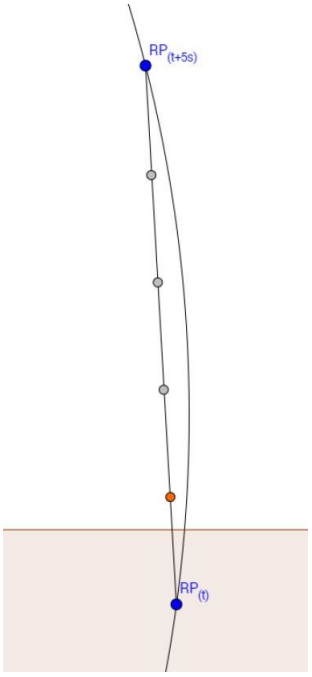
Die Übertrittkoordinate kann beim Wechsel von Sektor A in Sektor B bestimmt werden, indem die Strecke zwischen R2 und R3 mit der gemeinsamen Sektorgrenze der Sektoren geschnitten wird. Bei der Betrachtung der Abbildung 4.4(a) und 4.4(b) fällt jedoch auf, dass die Berechnung des Schnittpunktes ebenso ungenau ist. Der genaue Flugverlauf ist bei einer Kurve nicht bekannt. So muss auch hierbei von einer Ungenauigkeit ausgegangen werden. Es liegen keine Informationen über die Strecke zwischen den beiden bekannten Radarpositionen vor. Eine Bestimmung der exakten Flugbahn müsste über eine weitergehende Betrachtung der komplett geflogenen Strecke erfolgen. Im folgenden Abschnitt wird, trotz der Ungenauigkeit, der durch Schneiden berechnete Punkt als der exakte Austrittspunkt verwendet. Um die Berechnung des Schnittpunktes durchzuführen, muss jede Grenze jedes Airblocks des Sektors mit der Strecke zwischen den beiden bekannten Koordinaten geschnitten werden. Diese Berechnung ist für Sektoren, die aus mehreren Polygonen bestehen und viele Eckpunkte besitzen, sehr aufwendig. Alternativ kann die Austrittskoordinate näherungsweise bestimmt werden. Hierzu kann die Strecke zwischen zwei Radarpositionen segmentiert werden. Dies ist in Abbildung 4.4(c) dargestellt. Durch anschließendes Sektorzuweisen der erstellten interpolierten Koordinaten kann die Position des Sektoreintrittes weiter eingegrenzt werden. Hierzu werden auf der direkten Strecke zwischen beiden bekannten Koordinaten gleichmäßig vier Koordinaten interpoliert. Die erste interpolierte Koordinate, die im neuen Sektor liegt, ist die Koordinate des Sektoreintritts und -austritts. Durch das Teilen der Strecke in fünf Segmente kann die Genauigkeit des Sektoraustritts (exemplarisch bei einem Airbus A320) auf theoretische 240 m (1200 m / 5) verbessert werden. Die Genauigkeit wird relativ zum berechneten Schnittpunkt ausgedrückt. Die Genauigkeit der Berechnungen ist in Abbildung 4.4(d) dargestellt.



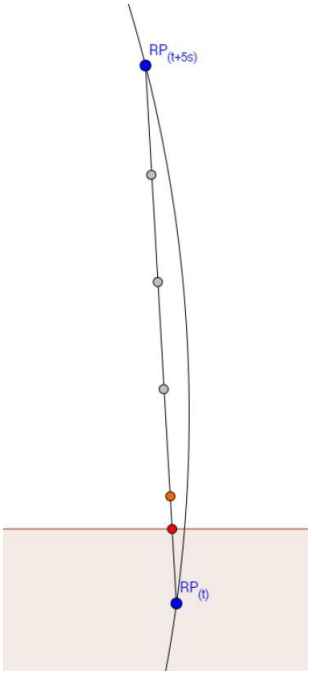
(a) Flugstrecke zwischen zwei Koordinaten



(b) Berechneter Austrittspunkt des Fluges aus einem Sektor



(c) Segmentierte Flugstrecke mit erstem Ausgetretenen Punkt



(d) Vergleich der Austrittspunkte beider Verfahren

Abbildung 4.4: *Probleme bei der Genauigkeit bei Sektorein- und Sektoraustritten*

Vor der Sektorzuordnung der neuen interpolierten Koordinate sind die OctreeElemente O_1 und O_2 der ursprünglichen zwei Radarkoordinaten bekannt. Die Suche nach der zugehörigen OctElemente der interpolierten Koordinaten – und damit dem Sektor – muss nur in dem gemeinsamen Parent-Element der OctreeElemente O_1 und O_2 durchgeführt werden. Die Suche des Sektors zu einer interpolierten Koordinate verläuft damit in weniger Suchschritten innerhalb des Octrees, als zu einer beliebigen Radarkoordinate.

	Ausgangsdaten	Segmentierung	Berechnung
Genauigkeit in Sekunden	5	1	0
Dauer in ms	4.702	5.414	12.564

Tabelle 4.3: *Laufzeit der Statistikerstellung mit verschiedenen Genauigkeiten*[6]

Tabelle 4.3 zeigt die Laufzeiten des Statistikmoduls mit den drei beschriebenen Genauigkeiten. Werden die Fünfsekundenintervalle beibehalten, so benötigt die Statistikerstellung etwa 4,7 Sekunden. Durch die Segmentierung verlängert sich die Statistikerstellung um 700 Millisekunden auf 5,4 Sekunden. Innerhalb der Tagesauswertung für den 22.05.2015 werden 256.400 Koordinaten durch Interpolation erstellt und ausgewertet. Das Erzeugen von interpolierten Koordinaten ist nur dann notwendig, wenn ein Sektorwechsel erkannt wird. Wird der Schnittpunkt anstatt der berechneten Segmente verwendet, wird die Statistikerstellung von 5,4 Sekunden auf 12,5 Sekunden verlängert.

Die Genauigkeit der Statistiken ist besonders im An- und Abflug relevant [LIZ17]. Hier treten viele Sektorwechsel und viele Kurven auf. Flugzeuge fliegen in diesen Flugphasen keine Reisegeschwindigkeit [Bre17]. Bei geringerer Geschwindigkeit ist die Bestimmung der Austrittskoordinate mittels Segmentierung genauer. Durch die Laufzeitvorteile wird in der Implementierung des Statistikmoduls die Segmentierungsmethode verwendet. Hierbei wird von der geringfügigen Ungenauigkeit abgesehen.

[6] Die Genauigkeit der Segmentierung ist berechnet

4.2.3 Streckenberechnung

Eine Kennzahl, die innerhalb der Statistiken zusätzlich berechnet werden soll, ist die zurückgelegte Strecke eines Luftfahrzeuges. Die Strecke soll auf Sektorebene dokumentiert werden. Zu jedem Sektoraufenthalt zwischen dem Ein- und Austritt aus einem Sektor soll dokumentiert werden, welche Strecke ein Flug innerhalb des Sektors zurückgelegt hat. Hieraus lässt sich auch eine deutschlandweite Statistik berechnen. Wichtig hierbei ist, dass die zurückgelegte Strecke nicht die Luftlinie zwischen der Sektorein- und -austrittskoordinate ist (in Abbildung 4.5 rot markiert). Es müssen alle einzelnen Streckensegmente zwischen den ausgewerteten Radarpositionen ausgewertet werden. Dies wird in Abbildung 4.5 verdeutlicht. Diese Strecke soll in Kilometern angegeben werden.

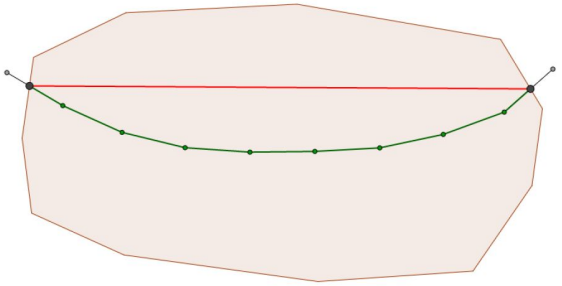


Abbildung 4.5: *Zurückgelegte Distanz innerhalb eines Sektors*

Zwischen zwei Koordinaten bewegt sich der Flug auf einer Kugeloberfläche. Eine Strecke auf dem Großkreis zwischen zwei Lat/Long-Koordinaten kann über die Haversine Formel berechnet werden (siehe [AMAA16]). Die Implementierung der Haversine-Formel liegt bereits vor und muss nicht entwickelt werden.

4.3 Testen des implementierten Statistikmoduls

Um die Funktionalität des Systems zu bewerten, wurde eine Grob- und Feinanalyse durchgeführt. Final wird das Modul während des Betriebs getestet. Hier laufen das STANLY-System und das Statistikmodul synchron und können kontinuierlich getestet werden.

4.3.1 Grobanalyse

Bei der Grobanalyse werden zunächst einzelne Flüge mit dem jeweiligen Datensatz des Altsystems verglichen. Es wurden sieben Flüge mit STANLY und dem Statistikmodul ausgewertet. Im ersten Schritt wird jeweils der Zeitpunkt jedes Sektorwechsels in beiden Systemen verglichen. Anschließend werden die Abweichungen der Zeitpunkte bestimmt. Hierzu liegen die Daten des 22.05.2015 vor.

Abweichung des Sektorwechsels (STANLY zu Statistikmodul)	Anzahl
0-4 s	22
5-10 s	5
11-20 s	6
Gesamt	33

Tabelle 4.4: *Klassifizierung der getesteten Abweichungen der Grobanalyse*

Innerhalb dieser Betrachtung gibt es 33 Sektorwechsel. 22 der untersuchten Sektorwechsel hatten eine zeitliche Abweichung von 0-4 Sekunden (siehe Tabelle 4.4). Diese Unterschiede resultieren aus der Segmentierung eines Sektorwechsels. Durch das Zerteilen der fünf Sekunden langen Radarstrecken ist der Zeitpunkt des Sektorwechsels um bis zu vier Sekunden versetzt. Die größeren Abweichungen sollen in einer Feinanalyse näher betrachtet werden.
Die berechneten Sektorein- und Sektoraustrittskoordinaten sowie die zurückgelegte Strecke können nicht mit dem STANLY-System verglichen werden. Es bietet keine Funktion diese Daten

zu erheben. Eine grobe Analyse der Koordinaten kann durchgeführt werden, indem geprüft wird, ob die berechneten Koordinaten auf der Strecke zwischen den bekannten Koordinaten liegt und damit korrekt berechnet wurden. Es traten während dieses Tests keine Unregelmäßigkeiten auf. Die Berechnung der Strecke zwischen zwei Koordinaten mittels Haversine Formel ist nicht notwendig. Es muss geprüft werden, ob alle Strecken innerhalb eines Sektors berücksichtigt wurden. Hierzu werden die einzelnen Koordinaten eines Sektordurchfluges exportiert. Diese Liste wird anschließend ebenfalls mit der Haversine-Formel berechnet und mit den Werten des Statistikmoduls verglichen. Hier ergeben sich keine Abweichungen.

4.3.2 Feinanalyse

Innerhalb der Feinanalyse wurden 601.175 Koordinaten ausgewertet. Diese stammen aus dem STANLY-System und sind je einem Sektor und je einem Flug zugeordnet. Ziel der Feinanalyse ist es, die Zahl der Abweichungen des Moduls zum Altsystem zu kennen. Bei der Statistikerstellung wurden 23.262 (3,9 %) Unterschiede bei einzelnen Koordinaten und dabei beim Sektorwechsel erkannt. Die Unterschiede wurden analysiert und kategorisiert.

Fehlerursache	Anzahl
Falscher Sektor in STANLY	14.703
Fehler durch Wiederholung	6.183
Koordinate liegt nicht innerhalb der Karte	2.367
Gesamt	23.262

Tabelle 4.5: *Kategorisierung der Unterschiede bei der Feinanalyse zwischen STANLY und dem entwickelten Statistikmodul*

Der korrekte Sektor für die Sektorbestimmung wird im Zweifelsfall mit der Bruteforce-Methode bestimmt. Hierbei werden alle bekannten Sektoren – ohne Octree – auf die gesuchte Koordinate geprüft.
6.183 Datensätze der Quelldaten aus dem STANLY-System sind widersprüchlich. Hier wird eine

identische Koordinate zweifach aufgeführt und ist zwei unterschiedlichen Sektoren zugeordnet. Einer der beiden STANLY-Datensätze ist damit nicht korrekt. Um den korrekten Sektor und Datensatz zu ermitteln, werden beide mittels Bruteforce überprüft. Die Ergebnisse des Statistikmoduls entsprachen in jedem dieser Fälle dem korrekt ermittelten Datensatz.

In 14.703 Fällen zeigen das Statistikmodul und STANLY verschiedene Sektoren an. Auch hier wurde mittels Bruteforce bestimmt welche Methode richtig liegt. Hier konnte in jedem Fall nachgewiesen werden, dass der Octree den richtigen Sektor zurückgegeben hat. STANLY hingegen lag in keinem Fall korrekt. Bei ca. 35 % der 14.703 Fälle liegt die Höhe der Flugkoordinate eindeutig nicht innerhalb des Sektors den STANLY zurückgibt. Bei diesen Fällen kann ein Fehler durch unterschiedliche Projektion ausgeschlossen werden.

In 2.367 Fällen konnte innerhalb des Statistikmoduls kein Sektor ermittelt werden. STANLY hingegen ermittelt zu diesen Koordinaten einen Sektor innerhalb des durch die DFS kontrollierten Luftraumes. Auch die Bruteforce-Methode konnte keine der Koordinaten einem Sektor zuweisen.

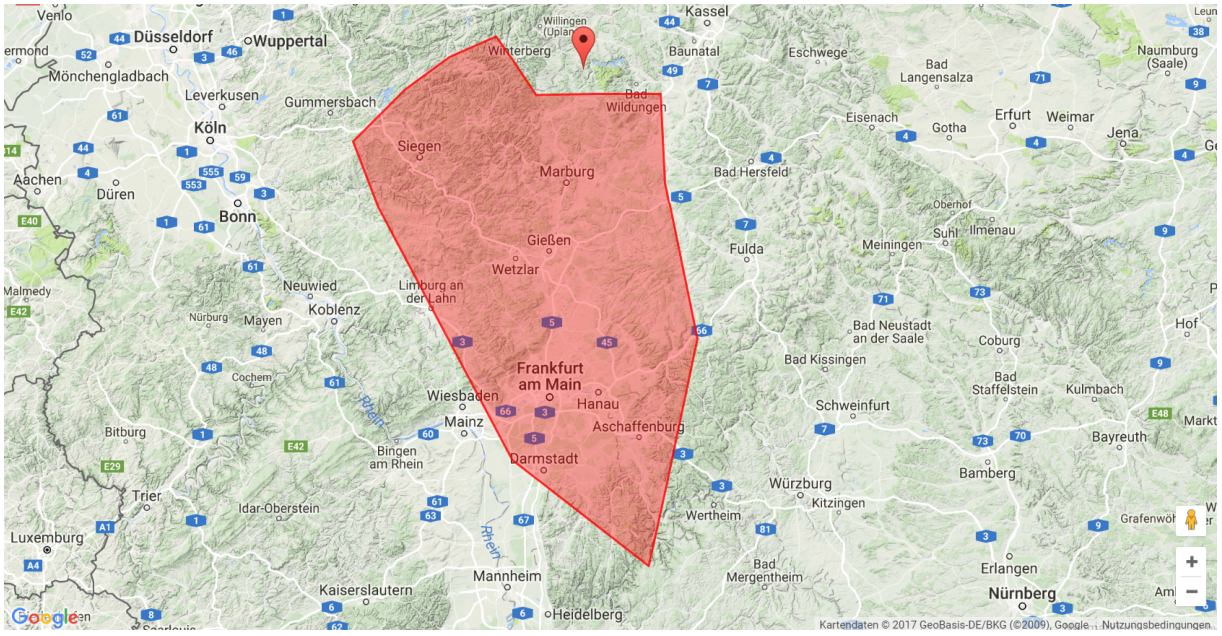


Abbildung 4.6: *Fehlerdarstellung einer Koordinate mit dem STANLY-Sektor (EDUUFFM23) innerhalb der Google Maps JavaScript API*

Um die Unterschiede zu veranschaulichen und stichprobenartig zu validieren wurde die Google Maps JavaScript API verwendet [goo17]. Hier wurden HTML-Dateien exportiert. Innerhalb dieser Dateien wird mit JavaScript der Sektor, in dem die gesuchte Koordinate sich laut STANLY befindet, abgebildet. Zusätzlich dazu wird die Koordinate dargestellt. Ein Beispiel dazu ist in Abbildung 4.6 dargestellt. Alle verwendeten Eckkoordinaten, sowie die gesuchte Koordinate werden in Lat/Long-Werten angegeben. Hier werden die Werte aus den Quellen verwendet und durch die Google-Maps-API projiziert. Dies führt auch zu einem Ausschluss von Projektionsfehlern. Die getesteten Fälle weisen analog zu den vorherigen Tests nur Fehler im STANLY-Ergebnis auf.
Abschließend lässt sich der Unterschied in einigen Punkten auf ein Feature von STANLY zurückführen. Hierbei werden Sektorein- und Sektoraustritte nur dann gezählt, wenn sich der Flug mindestens 60 Sekunden innerhalb des neuen Sektors befindet. Wenn ein Flug beispielsweise auf einer Sektorgrenze fliegt oder einen Sektor nur touchiert, werden die Sektorwechsel nicht dokumentiert. Bei der Analyse der Tagesstatistiken vom 22.05.2015 zeigt sich, dass rund 15 % der vom Statistikmodul ausgewerteten Flüge sich in mindestens einem Sektor für weniger als 20 Sekunden aufhalten. Deshalb muss die Methodik von STANLY in das Statistikmodul überführt werden. Die Vernachlässigung dieser Schwellenzeit führt zu einer unbrauchbaren Statistik mit hohen Ein- und Austrittszahlen. Ob hierbei auch von 60 Sekunden ausgegangen wird, muss noch betrachtet werden.

4.4 Fazit

Aus diesem Abschnitt geht hervor, dass die Daten aus STANLY nur bedingt mit dem neuen Statistikmodul verglichen werden können. Ergebnisse, die sich unterscheiden, benötigen eine aufwendige Einzelprüfung. Stichprobenartig wurden diese durchgeführt. Die Unterschiede beider Systeme können auf zwei Eigenschaften von STANLY und des Statistikmoduls zurückgeführt werden. Zum einen unterscheiden sich die Projektionen beider Systeme. Hier kann eine Differenz beim Umwandeln der Koordinaten entstehen. Zudem verwendet STANLY die

beschriebene Filterzeit von 60 Sekunden. Das neue Modul verfügt noch nicht über eine ähnliche Methodik.

Die Erstellung der Tagesstatistik – für den 22.05.2015 – mit dem in diesem Abschnitt erläuterten Statistikmodul wird in ca. 5,4 Sekunden erzeugt (exkl. des Einlesens der Radardaten). Innerhalb von 1,5 Sekunden werden sowohl die eventbasierten Statistiken als auch die Auslastungsstatistiken in CSV-Dateien exportiert. Zunächst wird der CSV-Export nicht durch eine Datenanbindung ersetzt. Innerhalb der EDIS-Struktur muss erst eine MySQL-Umgebung zur Verfügung gestellt werden. EDIS erzeugt für jeden Tag eine CSV-Datei, mit allen Radarpositionen. Die verwendete Testdatei ist eine Tagesdatei, die von EDIS erzeugt wurde. Nach der ersten Integration wertet EDIS zunächst nicht den Live-Radardatenstrom aus, sondern am Ende des Tages die erstellte CSV-Datei.

5 Integration

Im folgenden Abschnitt wird erläutert, welche Schritte betrachtet werden müssen, um das Statistikmodul und die Octree-Implementierung in EDIS zu integrieren. Wie bereits in Abschnitt 4.4 angedeutet wurde, ist die Umgebung von EDIS noch nicht vollständig vorbereitet. Trotzdem wurden die Octree-Datenstruktur und das Statistikmodul zunächst prototypisch in ein EDIS-Abbild integriert. Hierbei wird der Octree, sowie das Statistikmodul in je ein Paket exportiert.

5.1 Apache Camel

Apache Camel ist ein Java-basierter Routing-Engine-Builder. Hierbei bietet Camel die Möglichkeit verschiedenste Routing-Regeln zu definieren [IA11]. Im LIZ laufen über die in EDIS integrierten Camel-Routen unter anderem die Radardaten der verschiedenen DFS-Systeme. Diese werden über Apache ActiveMQ an weitere Systeme verteilt. AcitveMQ ist ein Message-Broker der Apache Organisation [apa11].
Erstellte Routingregeln können zunächst über die Parameter „from“ und „to“ spezifiziert werden. Diese definieren die Datenquelle und die Datensenke der zu verarbeitenden Daten. Neben diversen weiteren Parametern, können bei der Routenerstellung einzelne Datenverarbeitungsschritte in „process“-Routinen abgearbeitet werden. In dem vorliegenden Anwendungsfall werden in

EDIS Radardatenstrings (siehe Code 5.1) über ein ActiveMQ-Topic versendet. Hier kann der eingetroffene Radardatensatz im Octreemodul einem Sektor zugewiesen werden. Der zugewiesene Sektor wird anschließend mittels „processor“ vor der weiteren Verarbeitung an den empfangenen Radardatensatz angehängt. Dieses Vorgehen ist in Code 5.2 abgebildet.

```
#ID;            callsign; Date;
2015052200013; TUI7GM;    2015.05.22 00:00:34.734;

#X-Koordinate;        Y-Koordinate;      Höhe;
-220.44650869757322;  5.263712776529337; 42.0;
```

Quellcode 5.1: *Radardatenstring*

```
public class RadarRoute extends RouteBuilder {
    private OctElement root;
    public static class ProcessorRoute {
        public void processLine(Exchange exchange) throws Exception {
            //Nachricht speichern und an ";" splitten
            String line = exchange.getIn().getBody(String.class);
            String [] parts = line.split(";");
            // Sektor zur Position des Datensatzes finden
            SearchResult sr = root.searchForPoint(
                    new Point2D.Double(Double.parseDouble(parts[2]), Double.parseDouble(
                        parts[3])),
                    Double.parseDouble(parts[4]),
                    true);
            //Sektor an den Radardatenstring hängen
            line += sr.getSektor().getName()+";";
            //Output-Message manipulieren
            exchange.getOut().setBody(line);
            exchange.getOut().setHeaders(exchange.getIn().getHeaders());
        }
    }
```

```
20      @Override
21      public void configure() throws Exception {
22
23          //Octree aus einer Datei wiederherstellen
24          root = initPolygons.importOctreeFromFile("fuparOctree.export", true, DatenTyp.
                Pfad).root;
25
26          //Processor für die Route
27          ProcessorRoute processor = new ProcessorRoute();
28
29          /* von Topic ''activemq:topic:trackstring'' nach
30          * ''activemq:topic: tracksektorstring ''
31          * Input-Daten im o.g. CSV-Format
32          * Processor-Methode: processLine
33          */
34          from("activemq:topic:trackstring")
35          .id("RADAR_POINT_INPUT")
36          .bean(processor,"processLine")
37          .to("activemq:topic:tracksektorstring");
38      }
39  }
```

Quellcode 5.2: *Quellcode zur Erweiterung der ActiveMQ Radarstrings um einen Sektor*

Das gezeigte Octreemodul subscribed das ActiveMQ-Topic „activemq:topic:trackstring“ innerhalb von EDIS. Die Daten werden anschließend verarbeitet, ergänzt und schließlich in einem anderen ActiveMQ-Topic gepublished. So können auch andere Systeme auf die Sektorzuweisung des Octrees zugreifen.

Das Statistikmodul wird später die ActiveMQ Nachrichten des beschriebenen Octrees auswerten. Es erhält hierfür auch eine ActiveMQ-Route mit entsprechendem Processor. Das Statistikmodul ist zunächst – bis zur Anbindung an eine Datenbank – offline betrieben. Hierfür wird eine Route implementiert, die alle CSV-Dateien mit Radarkoordinaten innerhalb

eines bestimmten Ordners öffnet. Aus der dabei eingelesenen Datei werden anschließend die Statistiken erzeugt. Die Tageskoordinatensätze – im CSV-Format – werden innerhalb von EDIS bereits täglich erstellt und nun zusätzlich in den Ordner des Statistikmodules gespeichert.

5.2 Apache Karaf Struktur

Apache Karaf bildet den Kern von EDIS. Karaf ist ein Container, in dem verschiedene Java-Anwendungen zur Laufzeit installiert, gestartet und gestoppt werden können. Hier werden die installierten Anwendungen mittels Apache Felix im OSGi-Framework implementiert. Diese Anwendungen werden als Bundles bezeichnet [apa16]. Das OSGi-Framework stellt eine dynamische Softwareplattform dar. Diese Plattform verwaltet Java-Komponenten modular [Fun09]. OSGi ist besonders durch seine Offenheit, Modularität und Erweiterbarkeit für EDIS geeignet. Die in Karaf installierten Bundles können auf die freigegebenen Packages und Klassen der anderen Bundles zugreifen [HK14]. Ein Vorteil dieser EDIS Struktur ist, dass Bundles unterschiedlicher Versionsnummern gleichzeitig betrieben werden können. Dies führt dazu, dass Änderungen in einem Bundle sich nicht auf andere Bundles auswirken.
Innerhalb des Anwendungsfalles werden drei Arten von Bundle-Kommunikation verwendet. Zum einen gibt es Bundles, die eine Apache Camel Route beinhalten. Diese Bundles erstellen Routen und verarbeiten sie. Zudem gibt es Bundles, die einen Service zur Verfügung stellen. Sie bieten ein Interface, über das eine andere Anwendung bzw. ein anderes Bundle auf die Funktion der Interface-Klasse zugreifen kann. Weitere Klassen und Packages neben denen der Interface-Klasse werden nicht zur Verfügung gestellt. Darüber hinaus kann auch ein Bundle installiert werden, das nur verwendet wird, um den anderen OSGi-Bundles verschiedene Klassen und Datentypen zur Verfügung zu stellen. Diese unterscheiden sich von den vorherigen darin, dass sie kein festes Interface und keinen Service, sondern ihre Klassen zur Verfügung stellen. Die Datenstruktur der Radarpositionen ist ein Beispiel für ein solches Bundle. Radarpositions-Objekte werden bspw. sowohl im Statistikmodul als auch im Octreemodul verwendet.
Wie bereits erläutert werden die Radardaten durch das Octreemodul mittels Camel Route bear-

beitet. Für die in Abschnitt 4.2.2 angesprochene Genauigkeit ist es notwendig, dass das Bundle des Octrees zusätzlich dazu einen Service zur Verfügung stellt. Koordinaten, die innerhalb des Statistikmoduls (siehe Abschnitt 4.2.2) berechnet werden, sollen anschließend über diesen Service einem Sektor zugewiesen werden. Das Statistikmodul wird später ebenfalls als Service betrieben. Im Interface des Statistikmoduls wird hierbei die Funktion addRadarKoordinate() zur Verfügung gestellt. Zum aktuellen Zeitpunkt wird das Statistikmodul jedoch vollständig in ein Bundle ohne weitere Funktionen integriert. Die Offline-Statistik wird schließlich in dem Modul erzeugt, in dem die Radarkoordinatendatei weiterverarbeitet wird. Dieses erstellt eine Instanz des Statistikmoduls und ruft für jede Koordinate die addRadarKoordinaten()-Methode auf. Die Datentypen, die von mehreren anderen Bundles verwendet werden sollen (bspw. Sektor, OctElement, etc.), werden ebenfalls in ein Bundle exportiert.

5.3 EDIS

Die beschriebenen OSGi Bundles wurden zunächst in ein Abbild von EDIS integriert. Hier wurden im Rahmen der vorliegenden Arbeit die Funktion und die Lauffähigkeit der Systeme überprüft und verbessert. Der nächste Schritt wird die Integration des Octree-Bundles in eine operative EDIS-Intanz sein. Das Statistikmodul wird in den folgenden Wochen ebenfalls integriert.

Danach wird eine MySQL-Datenbank an EDIS angebunden. Steht die Datenbank zur Verfügung, werden das Octree- und das Statistikmodul weiter angepasst. Hierbei wird das Octree-Modul durch die Octree-Factory ersetzt.

6 Schlussbetrachtung

6.1 Fazit

Die in einer vorherigen Arbeit [Ger16] entwickelte Octreestruktur wurde um eine schnelle Erzeugungsmethode erweitert. Diese verkürzt das Aufbauen eines Octrees für eine STANLY-Sektor-Karte von 41 Sekunden um 83 % auf sieben Sekunden. Trotz der verkürzten Aufbauzeit wurde auch der Export und Import weiterentwickelt. Er verfügt zusätzlich zum CSV-Export nun über eine OctreeFactory mit MySQL-Anbindung. Sie ermöglicht das Laden eines STANLY-Octrees in 3,2 Sekunden. Durch die Kompression der Exportdaten vor dem Speichern, sowie der Dekompression nach dem Lesen, konnte der Import zusätzlich auf 2,3 Sekunden verkürzt werden. Hierbei wurde der Exportstring von 70 MB auf 9 MB verkleinert. Das Speichern eines Octrees in der Datenbank benötigt unkomprimiert und komprimiert (inkl. der Komprimierung) 1,7 Sekunden.
Das Statistikmodul erstellt für die Ausgangsdaten eines Tages in 5,4 Sekunden sowohl eventbasierte Flugstatistiken als auch Sektorauslastungsstatistiken. Die eventbasierten Statistiken können in eine MySQL-Datenbank geschrieben werden. Diese wird über eine OpenJPA-Verbindung angesprochen. Das Speichern der ca. 200.000 Datensätze benötigt 32 Sekunden und wird in einer späteren Live-Auswertung über die Laufzeit eines Tages verteilt.
Das Statistikmodul wurde prototypisch modularisiert. Die Module wurden in je ein OSGi Bundle exportiert und in eine EDIS-Testumgebung überführt. Hier soll das Statistikmodul für alle

Radarkoordinaten jedes Tages eine Statistik erstellen. Da die EDIS-Umgebung noch nicht über eine MySQL-Datenbank verfügt, wird der Export der Statistiken und der Export des Octrees weiterhin über CSV-Dateien realisiert, bis die Anbindung an die Datenbank realisiert ist.

6.2 Ausblick

Die geprüften Module müssen nun von der verwendeten Test-Instanz auf eine Live-Instanz migriert werden. Sobald die MySQL-Datenbanken in EDIS zur Verfügung stehen, wird der Export auf die Datenbank-Lösung umgestellt. Gleichzeitig zur Umstellung auf Datenbanken, werden die Apache Camel Routen für eine Live-Auswertung des Radars umgesetzt. Hier wird eine schrittweise Speicherung über OpenJPA realisiert. Zudem wird im Zuge der Änderungen das Statistikmodul weiter modularisiert.
Während des Betriebes des Statistikmoduls in EDIS werden die vom Statistikmodul erzeugten Statistiken weiter mit den STANLY-Daten verglichen. Zusätzlich muss der zeitliche Schwellenwert für Sektorein- und Sektoraustritte im neuen Statistikmodul realisiert werden. Hierzu muss ebenfalls eine Zeit festgelegt werden.
Um die Statistiken weiter zu verbessern, ist eine Anbindung an die Flugpläne notwendig. Hier können Fluginformationen erweitert werden, um die Abfragen zu optimieren. Einzelne Flüge, wie z. B. nicht von der DFS kontrollierte Sichtflüge, können hierdurch von den Statistiken ausgeschlossen werden.

A Anhang

A.1 Datenexport

A.1.1 STANLY

```
STANLY
20161111000000
/#### NXT ####/
#EDGGADF1,000,135,PFUNCTU
%P1_EDFF,P1_EDMM,MADAP
N501710 E093720
N495839 E094133
N500031 E092955
N500014 E091439
N500004 E090549
N495524 E084605
N500409 E084118
N501441 E092624
N501710 E093720
#EDGGADF10,000,135,PFUNCTU
```

```
%P1_EDFF,P1_EDMM,MADAP
N502237 E083633
N501829 E083059
N501156 E083428
N501127 E083226
N500923 E082348
N501751 E081201
N501902 E081759
N501946 E082149
N502046 E082655
N502237 E083633
 [...]
/#### NXT ####/
−288.18190260207774;−469.8455218203054;
350.58212970663493;465.183297187315;
 0.0;660.0;
V;
VA;
VAA;
VAAA;
VAAAA;Sektorlos;
VAAAB;
 [...]
VAHDHEGF;
VAHDHEGFA;EDWWHRZ4;
VAHDHEGFB;EDWWHRZ4;
VAHDHEGFC;EDGGHEF10;EDWWHRZ4;
VAHDHEGFD;EDGGHEF10;EDWWHRZ4;
VAHDHEGFE;EDWWHRZ4;EDYYH3SL_1;
VAHDHEGFF;EDWWHRZ4;EDYYH3SL_1;
VAHDHEGFG;EDGGHEF10;EDWWHRZ4;EDYYH3SL_1;EDYYH3SL_2;
VAHDHEGFH;EDGGHEF10;EDWWHRZ4;EDYYH3SL_1;EDYYH3SL_2;
```

```
48 [...]
```

Quellcode A.1: *Datenexport eines STANLY-Octrees*

A.1.2 FRODO

```
1  FRODO
2  20160101010000
3  /#### NXT ####/
4  name,polygonname,hoeheunten,hoeheoben,ebname,ebgname,id,counter,xcoord,ycoord,punkt
5  EDGGADF:027ED:0:115,027ED,0,115,EDGGADF,"EBG-Mitte 10",931,0,50.38806,9.40417,
6  EDGGADF:027ED:0:115,027ED,0,115,EDGGADF,"EBG-Mitte 10",931,1,50.42361,9.59056,
7  EDGGADF:027ED:0:115,027ED,0,115,EDGGADF,"EBG-Mitte 10",931,2,50.45333,9.58361,
8  EDGGADF:027ED:0:115,027ED,0,115,EDGGADF,"EBG-Mitte 10",931,3,50.51083,9.48611,
9  EDGGADF:027ED:0:115,027ED,0,115,EDGGADF,"EBG-Mitte 10",931,4,50.52028,9.37083,
10 EDGGADF:191ED:0:135,191ED,0,135,EDGGADF,"EBG-Mitte 10",2897,0,50.04361,7.68667,
11 EDGGADF:191ED:0:135,191ED,0,135,EDGGADF,"EBG-Mitte 10",2897,1,49.94611,8.17611,
12 EDGGADF:191ED:0:135,191ED,0,135,EDGGADF,"EBG-Mitte 10",2897,2,49.90334,8.38750,
13 EDGGADF:191ED:0:135,191ED,0,135,EDGGADF,"EBG-Mitte 10",2897,3,49.88305,8.48667,
14 EDGGADF:191ED:0:135,191ED,0,135,EDGGADF,"EBG-Mitte 10",2897,4,49.87055,8.54722,
15 EDGGADF:191ED:0:135,191ED,0,135,EDGGADF,"EBG-Mitte 10",2897,5,49.92334,8.76805,
16 EDGGADF:191ED:0:135,191ED,0,135,EDGGADF,"EBG-Mitte 10",2897,6,50.00111,9.09694,
17 EDGGADF:191ED:0:135,191ED,0,135,EDGGADF,"EBG-Mitte 10",2897,7,50.00389,9.24417,
18  [...]
19 /#### NXT ####/
20 -523.2324833498938;-470.49245078969574;
21 348.906891055338;456.0938829782786;
22  0.0;500.0;
23 V;
24 VA;
25 VAA;Sektorlos;
```

```
26 VAB;
27 VABA;
28 VABAA;Sektorlos;
29 VABAB;
30  [...]
31 VAGFFFHH;
32 VAGFFFHHA;EDYYD4WL:031EH:245:345;
33 VAGFFFHHB;EDYYD4WL:031EH:245:345;
34 VAGFFFHHC;EDYYD4WL:031EH:245:345;
35 VAGFFFHHD;EDYYD4WL:031EH:245:345;EDYYD4WL:abcEH(032EH):245:345;
36 VAGFFFHHE;EDYYD4WL:031EH:245:345;
37 VAGFFFHHF;EDYYD4WL:031EH:245:345;
38 VAGFFFHHG;EDYYD4WL:031EH:245:345;
39 VAGFFFHHH;EDYYD4WL:031EH:245:345;EDYYD4WL:abcEH(032EH):245:345;
40  [...]
```

Quellcode A.2: *Datenexport eines FRODO-Octrees*

A.2 UML-Diagramm Octree

Das UML-Diagramm zeigt die UML-Klassenstruktur der implementierten Octree-Struktur aus [Ger16]:

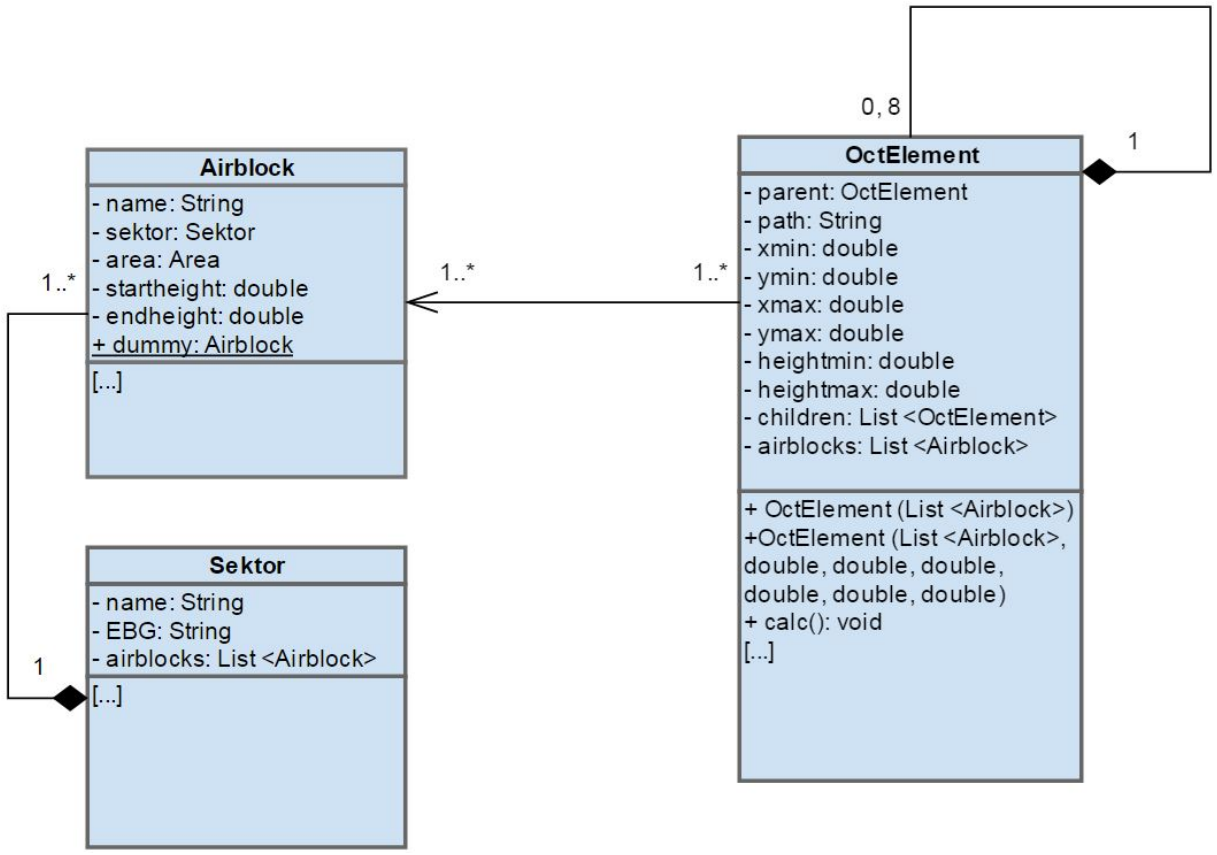


Abbildung A.1: *UML-Diagramm der Octree-Struktur*

Glossar

AIRAC	Aeronautical Information Regulation And Control: einheitlicher 28-tägiger von der ICAO festgelegter Zyklus, in welchem Luftfahrtunternehmen und -behörden Änderungen, wie bspw. Navigationspunkte, bekannt geben können
Airblock	Ein Airblock beschreibt einen Luftraum, mit einem festen Höhenbereich und einer Grundfläche
Apache ActiveMQ	Message Broker mit Publish-/Subscribe-Schema
Apache Camel	Java- und regelbasierte Routingengine
Apache Felix	Implementation des OSGi-Frameworks
Apache Karaf	Container & Laufzeitumgebung für bspw. OSGi-Bundles
Apache OpenJPA	JPA Provider
A320	Flugzeugtyp von Airbus
Blob	Binary Large Object
Callsign	Dt. Rufzeichen: Dient zur Identifikation der einzelnen Luftfahrzeuge im Luftverkehrsmanagement
CSV	Character-Separated Values: Datenreihen die durch ein beliebiges Trennzeichen getrennt sind

DFS	DFS Deutsche Flugsicherung GmbH
Docker	Software Container Plattform
EclipseLink	JPA Provider der Eclipse Foundation
EDIS	ESB des LIZ
ESB	Enterprise-Service-Bus: Stellt unterschiedliche Daten und Dienste innerhalb eines Unternehmens zur Verfügung
FRODO	System zur grafischen Darstellung von Lufträumen und Sektoren
Großkreis	Größtmöglicher Kreis auf einer Kugel (z. B. zwischen zwei Koordinaten auf der Erde)
ICAO	International Civil Aviation Organization
JDBC	Java Database Connectivity: Einheitliche Schnittstelle zur Kommunikation mit relationalen Datenbanken
JPA	Java persistence API: Einheitliche Schnittstelle für ORM mit Java-Objekt
Lat/Long-Koordinate	Kugelkoordinaten bestehend aus Breiten- und Längengrad
LIZ	Lage- und Informationszentrum der DFS
NoSQL	No SQL oder Not Only SQL: Nicht-Relationales Datenbankschema
Oracle NoSQL	Nicht-Relationale NoSQL Datenbank von Oracle
Oracle MySQL	Relationale Datenbank von Oracle

ORM	Object-relational Mapping: Speichern von Klassen-Objekten in Tabellen in Datenbanken
OSGi-Framework	Framework der Open Services Gateway Initiative: Framework zum Verwalten von Java-Anwendungen und Diensten
Sektor	Luftraum der sowohl horizontal wie auch vertikal begrenzt ist und von je zwei Lotsen kontrolliert wird. Ein Sektor kann sich aus mehreren Airblocks zusammensetzen
STANLY	eigentlich STANLY 3: Statistik- und Auswertesoftware innerhalb des LIZ
SQL	Structured Query Language: Einheitliche Sprache zum Abfragen von Relationalen Datenbanken
UML	Unified Modeling Language: Einheitliche Modellierungssprache bspw. zur Darstellung von Klassenstrukturen
VM	Virtuelle Maschine

Literatur

[AMAA16] Alam, C. N. ; Manaf, K. ; Atmadja, A. R. ; Aurum, D. K.: Implementation of haversine formula for counting event visitor in the radius based on Android application. In: *2016 4th International Conference on Cyber and IT Service Management*, 2016, S. 1–6

[apa11] Apache Software Foundation: *Apache ActiveMQ – Index*. `http://activemq.apache.org/`. Version: 2011. – Abgerufen am 20.02.2017

[apa15] Apache Software Foundation: *Overview*. `http://camel.apache.org/`. Version: 2015. – Abgerufen am 07.02.2017

[apa16] Apache Software Foundation: *Overview*. `https://karaf.apache.org/manual/latest/overview.html`. Version: 2016. – Abgerufen am 07.02.2017

[AR16] Anand, V. ; Rao, C. M.: MongoDB and Oracle NoSQL: A technical critique for design decisions. In: *2016 International Conference on Emerging Trends in Engineering, Technology and Science (ICETETS)*, 2016, S. 1–4

[BRA12] Boicea, A. ; Radulescu, F. ; Agapin, L. I.: MongoDB vs Oracle – Database Comparison. In: *2012 Third International Conference on Emerging Intelligent Data and Web Technologies*, 2012, S. 330–335

[Bre17] Bredow, Wolfgang: *Airbus A320-200: Die Variante Airbus A-320-200 zählt zur A320-Familie*. `http://www.bredow-web.de/Berlin_Tegel/Airbus_A320-200/Airbus_a320-200.html`. Version: 2017. – Abgerufen am 24.01.2017

[CGK15] Chickerur, S. ; Goudar, A. ; Kinnerkar, A.: Comparison of Relational Database with Document-Oriented Database (MongoDB) for Big Data Applications. In: *2015 8th International Conference on Advanced Software Engineering Its Applications (ASEA)*, 2015, S. 41–47

[dfs16] DFS Situation and Information Centre (LIZ): *Air traffic statistics : Annual Summary 2016*. `https://www.dfs.de/dfs_homepage/de/Unternehmen/Zahlen%20und%20Daten/Statistiken/Annual_Summary_2016.pdf`. Version: 2016. – Abgerufen am 12.01.2017

[dfs17] DFS Deutsche Flugsicherung GmbH: *DFS Deutsche Flugsicherung GmbH : Einführung in das Unternehmen*. 2017. – Präsentation des Unternehmens

[Fun09] Funke, Holger: *Das OSGi Framework*. `https://jaxenter.de/das-osgi-framework-8984`. Version: 2009. – Abgerufen am 23.02.2017

[Ger16] Gerome, Philipp: Backend-Processing von Radardaten zur Sektorbestimmung / DFS Deutsche Flugsicherung GmbH. 2016. – Forschungsbericht. – vertraulich

[GGPO15] Győrödi, C. ; Győrödi, R. ; Pecherle, G. ; Olah, A.: A comparative study: MongoDB vs. MySQL. In: *2015 13th International Conference on Engineering of Modern Electric Systems (EMES)*, 2015, S. 1–6

[goo17] Google Inc.: *Google Maps JavaScript API | Google Developers*. `https://developers.google.com/maps/documentation/javascript/?hl=de`. Version: 2017. – Abgerufen am 06.02.2017

[Had13] Hadjigeorgiou, Christoforos: *RDBMS vs NoSQL: Performance and Scaling Comparison*. Edinburgh, UK, 2013. – Master Thesis

[Hah13] Hahn, Michael: *Approach and Realization of a Multi-tenant Service Composition Engine*, University of Stuttgart - Institute of Architecture of Application Systems, Diplomarbeit, 2013. – S. 24 & S. 26

[Hau10] Hauer, Philipp: *Das Decorator Design Pattern*. https://www.philipphauer.de/study/se/design-pattern/decorator.php. Version: 2010. – Abgerufen am 20.02.2017

[HK14] Hackbarth, K. ; Kersten, K.: *OSGi | Heise Developer*. https://www.heise.de/developer/artikel/Java-und-OSGi-in-Embedded-Systemen-fuer-das-Internet-der-Dinge-2152241.html?artikelseite=2. Version: 2014. – Abgerufen am 20.02.2017

[Hua12] Huang, Rui: Optimizing collision detection in 3D games with model attribute and Bounding Boxes. In: *2012 IEEE Symposium on Electrical Electronics Engineering (EEESYM)*, 2012, S. 589–591

[IA11] Ibsen, C. ; Anstey, J.: *Camel in Action*. Manning, 2011 (Manning Pubs Co Series). – ISBN 9781935182368

[ica17] International Civil Aviation Organization (ICAO): *AIRAC*. http://www.icao.int/safety/information-management/Pages/AIRACAdherence.aspx. Version: 2017. – Abgerufen am 02.02.2017

[LIZ17] LIZ - DFS Deutsche Flugsicherung GmbH: *LIZ - DFS Deutsche Flugsicherung GmbH : Interne Informationen*. 2017

[Luf17] Deutsche Lufthansa AG: *Airbus A320-200 - Kontinental-Flotte - Lufthansa-Group*. https://www.lufthansagroup.com/unternehmen/flotte/lufthansa-passage/kontinental-flotte/airbus-a320-200.html. Version: 2017. – Abgerufen am 18.01.2017

[LZ10] Lai, Y. ; ZhongZhi, S.: An Efficient Data Mining Framework on Hadoop using Java Persistence API. In: *2010 10th IEEE International Conference on Computer and Information Technology*, 2010, S. 203–209

[ML06] Maciej Liskiewicz, Henning F.: *Datenkompression*. https://www.uni-trier.de/fileadmin/fb4/prof/INF/TIN/Folien/DK/script.pdf. Version: 2006. – Abgerufen am 27.01.2017

[NSM+12] N.Jatana ; S.Puri ; M.Ahuja ; I.Kathuria ; D.Gosain: A Survey and Comparison of Relational and Non-Relational Database / Maharaja Surajmal Institute of Technology. International Journal of Engineering Research & Technology (IJERT), 2012. – Forschungsbericht

[Ora14] Oracle Corporation: *Oracle NoSQL Database Large Object API*. `https://docs.oracle.com/cd/E57769_01/html/booklets/lobs/index.html`. Version: 2014. – Abgerufen am 27.01.2017

[ora16] Oracle Corporation: *Java JDK Source 1.8.0 101*. 2016. – Abgerufen am 17.01.2017

[Ora17a] Oracle Corporation: *MySQL :: MySQL 5.7 Reference Manual :: 12.8 Data Type Storage Requirements*. `https://dev.mysql.com/doc/refman/5.7/en/storage-requirements.html`. Version: 2017. – Abgerufen am 27.01.2017

[Ora17b] Oracle Corporation: *Oracle NoSQL Database Technical Overview*. `http://www.oracle.com/technetwork/database/database-technologies/nosqldb/overview/index.html`. Version: 2017. – Abgerufen am 12.01.2017

[Sal07] Salomon, David: *Data Compression: The Complete Reference*. Springer-Verlag London, 2007. – ISBN 978–1–84628–603–2

[Šem12] Šembera, Lukáš: *Comparison of JPA providers and issues with migration*, Masarykova Univerzita - Fakulta Informatiky, Diplomarbeit, 2012

[Sos99] Sosna, Dieter: *Sosna: Stereographische Projektion*. `https://www.informatik.uni-leipzig.de/~sosna/karten/stereograph.html`. Version: 1999. – Abgerufen am 22.11.2016

[Weg12] Wegrzynowicz, Patrycja: A Performance Comparison of JPA Providers. Poznan, Poland : GeeCON 2012, 2012

[XLC16] Xiaoyan, Q. ; Ling, M. ; Chengzhu, Y.: Research of collision detection algorithm based on hybrid bounding box for complex environment. In: *2016 International Conference on Integrated Circuits and Microsystems (ICICM)*, 2016, S. 248–252