# Semesterprojekt - Formale Beschreibung der Middleware

December 3, 2015

## 1 Index

### 1.1 Anforderungen

Der Index soll eine effiziente Suche nach Mutationen für das Frontend ermöglichen. Während des Programmstarts wird der Index aus der vorhandenen Datenbank aufgebaut und steht dann so lange zur Verfügung, bis das Programm beendet wird. Im Index wird mit Intervallgrenzen gesucht und der Index gibt alle Intervalle zurück, die komplett innerhalb des angegebenen Intervalls liegen.
Der Index wird verteilt aufgebaut und liegt auf 4 virtuellen Maschinen verteilt. Die für den Index spezifizierten Funktionen sprechen immer einen Teilindex an. Der Aufruf der Funktionen wird über den IndexController erfolgen, der immer alle 4 Teilindizes ansprechen wird.

### 1.2 Funktionen und Datenstrukturen

Die Funktionen des Indexes variieren in ihrerem Ablauf je nach gewählter Indexstruktur. Momentan existieren 3 Varianten, die getestet werden. Es ist nicht ausgeschlossen, dass weitere Strukturen im Laufe der Entwicklung getestet werden. Da die Dauer des Indexaufbaus für den Endnutzer nicht relevant ist hängt die Auswahl der letztendlich genutzten Struktur lediglich von der Geschwindigkeit der Suchanfragen ab. Im folgenden werden die Such-und Einfüge-Operationen basierend auf den jeweiligen Indexstrukturen beschrieben

### 1.2.1 IntervallTree

In diesem Fall basiert die Indexstruktur auf einem Intervallbaum.
Hierfür wird die frei zugängliche Bibliothek "IntervallST.java" der Universität Princeton genutzt. Beide Funktionen haben hier lediglich die Aufgabe als Interface zu den zugehörigen Bibliotheksfunktionen zu dienen: contains() zur Suche und put() zum Einfügen.
Das Suchergebnis wird nach möglichen Filtern gefiltert. Bei n Intervallen und einer Such-Ergebnisliste der Größe m ergibt sich ein Komplexität von O(log n + m)
Die Bibliothek muss noch angepasst werden, damit das Einfügen von gleichen Intervallen möglich ist ohne, dass das zuerst eingefügte Intervall gelöscht wird.

```
search()
found intervalls = contains(intervall);
forall the elements in found intervalls do
    if element corresponds to specified filters then
        add element to answer list;
    end
    return answer list;
end
```

```
addMutation()
if mutation is already in index then
    return "index already contains mutation";
end
put();
return "added mutation";
```

### 1.2.2 Suchbaumbasierter Index

Da ein Großteil der Intervalle einstellig bzw sehr kurz sind bietet sich ein einfacher binärer Suchbaum als Datenstruktur an.
Dieser wird um einen Iterator erweitert, damit effizient eine Menge an Knoten ausgewählt werden kann.
Hierfür wird die Java-Klasse TreeMap verwendet, die eine Suche in logarithmischer Zeit ermöglicht. Die Ergebnismenge wird einmal durchlaufen, Mutationen deren Endpunkt außerhalb des gesuchten Intervalls liegen werden dabei entfernt und jeder Knoten wird nach den angegebenen Filtern gefiltert. Bei n Intervallen und einer Such-Ergebnismenge von m Intervallen ergibt sich eine Laufzeit von O(log n +m).
Auch in dieser Implementation dienen die Funktionen als Interface zu den jeweiligen Funktionen der genutzten Klasse: submap() zur Suche und put() zum Einfügen von Objekten).

```
search()
found intervalls = submap(intervall);
forall the elements in found intervalls do
    if element corresponds to specified filters then
        add element to answer list;
    end
    return answer list;
end
```

```
addMutation()
if mutation is already in index then
    return "index already contains mutation";
end
put();
return "added mutation";
```

### 1.2.3 Arraybasierter Index

Sollte die Anzahl der Mutationen groß genug sein, dass sie mit Integer-Variablen darstellbar ist, so bietet sich unter Umständen auch ein arraybasierter Index an. Dieser speichert alle Mutationen aufsteigend sortiert nach ihrem Anfangspunkt. Wird nun nach einem Intervall gesucht, so iteriert er über das Array beginnend bei der Mutation, deren Startwert noch im gesuchten Intervall liegt. Dabei wird für jede Mutation überprüft, ob ihr Endwert noch im gesuchten Intervall liegt. Ist dem so wird sie zur Ergenismenge hinzugefügt. Hierbei können auch direkt die Filter überprüft werden.
Es so lange iteriert, bis der Startwert aller folgenden Mutationen größer, als der vom Nutzer angegebene Endwert ist.
Falls Mutationen an der gleichen Stelle beginnen, so verschieben sich alle folgenden Mutationen in der Liste, da in aufeinanderfolgenden Zellen gleiche Startintervalle gespeichert werden müssen. Es muss also ermittelt werden, wo sich die erste im Intervall liegende Mutation befindet. Ein Verfahren hierfür wird noch ermittelt
Bei m Mutationen, deren Startwert sich im gesuchten Intervall befinden, liegt die Laufzeit bei O(m + $\epsilon$), wobei $\epsilon$ davon abhängt, wie die erste Mutation ermittelt wird. Es kann aber davon ausgegangen werden, dass $\epsilon$ einen geringen Anteil an der Laufzeit ausmachen wird.

```
search()
find index x of first mutation that lies in intervall;
while starting point of mutation at index x lies in search intervall do
    if endpoint of mutation at index x lies in search intervall and
    mutation corresponds to specified filters then
        add mutation to answer list;
    end
    x=x+1;
end
return answer list;
```

```
addMutation()
if mutation is already in index then
    return "index already contains mutation";
end
insert mutation at corresponding index and adjust array properly;
return "added mutation";
```

# 2 IndexController

## 2.1 Anforderungen

Der IndexController nimmt Suchanfragen entgegen, leitet diese an die 4 Teilindizes weiter und fügt die Teilergebnisse wieder zusammen.
Falls eine Anfrage intern in mehrere Teilanfragen aufgeteilt werden sollte, weil z.B. ein Gen, nach dem gesucht wird an mehreren Stellen auftreten kann, leutet der IndexController alle Teilanfragen sequentiell an die Indizes weiter, fügt die Teilergebnisse zusammen und schickt die Ergebnismenge an den QueryReceiver zurück.

## 2.2 Funktionen

### 2.2.1 answerQuery(int[] intervals,String[] Sources,int[]filter)

Die Funktion erhält mehrere Listen als Parameter, die die nötigen Informationen für die einzelnen Anfragen beinhalten. Jeweils 2 aufeinanderfolgende Einträge in der Intervall-Liste beschreiben den Start-und Endpunkt der gesuchten Intervalle. Die Einträge in den anderen Listen werden für alle Anfragen genutzt
Die Anfragen werden sequentiell an die 4 Teilindizes weitergeleitet und einzelnen Ergebnisse konkateniert und in einer Liste zurückgegeben.

```
forall the queries in parameter list do
    forall the sub indices do
        answer list = index.search();
    end
    concatenate all answer lists;
end
concatenate answer lists of each sub query;
return answer list;
```

### 2.2.2 buildIndex()

Die Funktion wird bei Programmstart ausgeführt und baut auf Basis der Datenbank die 4 Teilinidzes auf.
Für jede Mutation wird zufällig entschieden in welchen Index sie eingefügt wird.

```
forall the elements in database do
    if element is already in index then
        skip element;
    end
    choose which sub index to insert into;
    index.addMutation();
end
return "index built";
```

# 3 Stresstest

Der Stresstest hat zum Ziel herauszufinden, wie lange die durchschnittliche Query-Laufzeit ist.
Ziel des Systems ist es eine Laufzeit von unter einer Sekunde zu erreichen.
Um dies zu überprüfen werden mehrere Anfragen der gleichen Art (Intervallsuche, Gennamenssuche, Präfixsuche) sequentiell gestellt und die Antwortzeit gemessen. Alle Anfragen sollen in unter einer Sekunde eine Antwort erzielen.