Institut für Formale Methoden der Informatik

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Grafisches Clustering von OSCAR-Suchresultaten

Sokol Makolli

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Stefan Funke
Betreuer/in:	Prof. Dr. Stefan Funke
Beginn am:	9. Juli 2018
Beendet am:	13. November 2018

Kurzfassung

Die Suchergebnisse der Suchmaschine OSCAR sind häufig für den Benutzer wegen ihrer großen Menge unübersichtlich. Eine Strukturierung auf Clientseite liefert zwar für den Nutzer relevante Ergebnisse, jedoch ist der dabei entstehende Kommunikationsoverhead zu groß und die Strukturierung somit zu langsam. In dieser Arbeit wird eine Strukturierung auf Serverseite vorgestellt, die Ergebnisse mit zehn- bis zwanzigfacher und bei einer regionalen Strukturierung mit bis zu hundertfacher Geschwindigkeit liefert.

Inhaltsverzeichnis

1 Einleitung

Wegen der freien Verfügbarkeit von geografischen Daten durch Open Street Map (OSM)[1] , sind zahlreiche Applikationen und Algorithmen entstanden, die diese großen Mengen an Daten aufbereiten und verarbeiten, um somit Probleme von Endnutzern zu lösen. Eine dieser Applikationen ist die von Bahrdt und Funke [BF15] entwickelte Suchmaschine OSCAR[2]. Diese erlaubt es, den gesamten OSM-Datensatz zu durchsuchen und Ergebnisse nach Lokalität oder durch die von OSM vorgegebenen Key-Value-Tags zu filtern.

Für das effektive Suchen muss der Endnutzer diese Key-Value-Tags kennen und in den Suchanfragentext eingeben, um damit die Ergebnismenge nach seinen Wünschen zu verfeinern. Alle chinesischen Restaurants in Stuttgart liefert zum Beispiel die Suchanfrage „@cuisine:chinese ”Stuttgart”“. Dabei ist „@cuisine:chinese“ ein Key-Value-Tag, wobei „cuisine“ der Key und „chinese“ das Value ist.

Grundsätzlich wird davon ausgegangen, dass der Benutzer diese Key-Value-Tags nicht kennt. Für das Benutzererlebnis wäre es somit von Vorteil, wenn eine sinnvolle Menge an Verfeinerungen dem Benutzer präsentiert werden, sodass er zum gewünschten Ergebnis kommt.

Für eine gute Benutzererfahrung ist es außerdem kritisch, die Strukturierung schnell zu berechnen, damit dieser an die Applikation gebunden wird und sie nicht verlässt. Kopf [Kop17] liefert eine Strukturierung der Ergebnismenge, die zwar sinnvolle Verfeinerungsmöglichkeiten für den Benutzer liefert, jedoch werden die Berechnungen auf dem Client durchgeführt. Die dadurch entstehende Übertragung der gesamten Ergebnismenge vom Server zum Client führt zu einem langsamen Algorithmus.

Somit ist der nächste Schritt, die Strukturierungsalgorithmen von Kopf [Kop17] eventuell zu verbessern und diese direkt an den OSCAR-Server anzubinden.

1.1 Aufgabenstellung

Die Aufgabe dieser Arbeit ist es eine lauffähige Erweiterung der OSCAR-Serverinfrastruktur in der Programmiersprache C++ zu programmieren, die möglichst schnell sinnvolle Verfeinerungsmöglichkeiten liefert. Dabei wurde als Referenz die Implementierung von Kopf [Kop17] gegeben. Außerdem soll eine Erweiterung der OSCAR-Weboberfläche implementiert werden, sodass der Benutzer die Ergebnisse vom Server interaktiv explorieren kann.

[1] https://www.openstreetmap.org
[2] http://www.oscar-web.de

1.2 Aufbau der Arbeit

Aus der Aufgabenstellung wird geschlossen, dass zunächst die OSCAR-Suchmaschine und die Strukturierung von Kopf [Kop17] analysiert werden muss, um dann eine Servererweiterung und Webseitenerweiterung zu implementieren.

Die Analyse findet in Kapitel 2 statt und die einzelnen Implementierungen werden in Kapitel 3 erläutert. In Kapitel 4 werden die Ergebnisse der Strukturierung an einigen Beispielsuchanfragen gezeigt und ein Geschwindigkeitsvergleich zwischen der Implementierung von Kopf [Kop17] und der Implementierung dieser Arbeit gezogen.

2 Relevante Arbeiten

Die Vorlage für diese Arbeit sind die Ergebnisse von Kopf [Kop17]. Diese werden in Abschnitt 2.2 weiter behandelt.

Kopf [Kop17] hat zudem als Inspiration für die Strukturierungsalgorithmen die von Ley [Ley02] entwickelte Suchmaschine für wissenschaftliche Publikationen DBLP[1] gewählt. In dieser werden Suchanfragen so strukturiert, wie es in Abbildung 2.1 zu sehen ist.

Es kann an der Strukturierung beobachtet werden, dass zum einen nach dem Key 'coauthor' und zum anderen nach dem Key 'venue' strukturiert wird. Es gibt in der Datenbank noch andere Keys nach denen strukturiert werden kann, jedoch erachtet DBLP es für am sinnvollsten die aufgeführte Strukturierung aufzuzeigen. Darunter werden die möglichen Values angezeigt, die nach der Anzahl der Suchergebnisse sortiert sind. Der Benutzer kann daraufhin eine dieser Values wählen und es wird nur nach Ergebnissen gesucht, die diesem Key-Value-Paar entsprechen. Die Strukturierung von Kopf [Kop17], und auch die dieser Arbeit, folgt diesem Model.

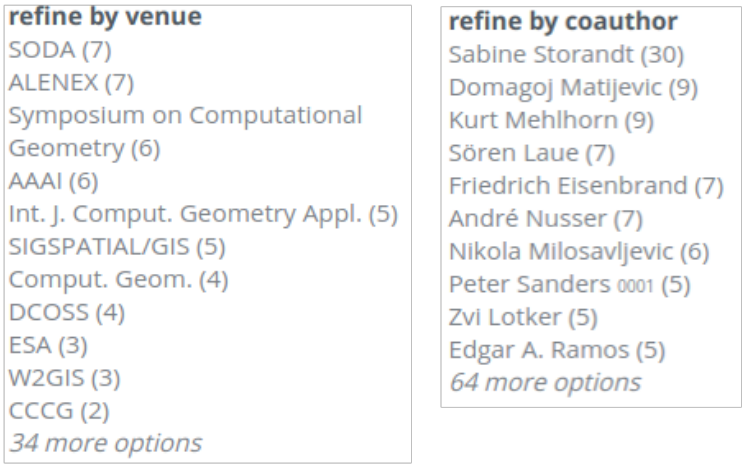

Abbildung 2.1: Ergebnisse der Strukturierung von DBLP auf die Suchanfrage Stefan Funke. Quelle: https://dblp.uni-trier.de/pers/hd/f/Funke:Stefan

2.1 Die OSCAR-Suchmaschine

Da diese Arbeit grundlegend auf dem OSCAR-Framework basiert, benötigt es, zum besseren Verständnis, einer Erklärung der Funktionsweise von OSCAR.

[1] https://dblp.uni-trier.de/

2.1.1 OSM-Grundlagen

OSCAR ist eine Sammlung an Komponenten die eine Textsuche auf den OpenStreetMap (OSM) Daten ermöglicht.

Die Daten von OSM bestehen aus sogenannten 'nodes' (Knoten), 'ways' (Wege), die eine zusammengefügte Anzahl an 'nodes' sind, und 'relations' (Relationen). Als Beispiel ist ein Ausschnitt der OSM-Datenbank in XML-Form in Listing 2.1 gegeben.

Administrative Einheiten werden mittels 'ways', 'relations' und 'nodes' dargestellt. Grenzen von Regionen sind geschlossene Polygone repräsentiert durch 'ways'. Alle 'nodes', 'ways' und 'relations' können mit Key-Value Tags ergänzt werden. [BF15]

Ein 'way' referenziert eine 'node' mit 'nd' und der 'node'-ID. Eine 'relation' hat verschiedene 'member' mit den Typen 'node' oder 'way' und der jeweiligen ID. Jede 'node' hat zusätzlich zu den Tags noch die Position in Längen- und Breitengraden gegeben.

Listing 2.1 Beispiel von OSM-Daten bestehend aus einem Knoten, einem Weg und einer Relation jeweils mit Tags. In Anlehnung an: https://wiki.openstreetmap.org/wiki/OSM_XML

```
<node id="1831881213" version="1" changeset="12370172" lat="54.0900666" lon="12.2539381"...>
  <tag k="name" v="Neu Broderstorf"/>
  <tag k="traffic_sign" v="city_limit"/>
</node>
<way id="26659127" user="Masch" uid="55988"...>
  <nd ref="292403538"/>
  <nd ref="298884289"/>
  ...
  <nd ref="261728686"/>
  <tag k="highway" v="unclassified"/>
</way>
<relation id="56688" user="kmvar"...>
  <member type="node" ref="294942404" role=""/>
  ...
  <member type="way" ref="4579143" role=""/>
  <tag k="name" v="Küstenbus Linie 123"/>
</relation>
```

2.1.2 OSCAR-Zellannordnung

OSCAR weist zum effizienten Suchen allen OSM-Regionen eine oder mehrere sogenannte Zellen zu (siehe Abbildung 2.2). Die Schnelligkeit, mit der die Suchergebnisse geliefert werden, wird dadurch gewährleistet, dass in der Praxis die Anzahl an Zellen signifikant niedriger ist als die Anzahl an Knoten. Die Knoten besitzen zusätzlich zu den eigenen Tags, auch die Tags der Regionen, in denen sie enthalten sind. Jede der Zellen wird dann mit den Tags aller in ihr enthaltenen Knoten versehen und die Suchstruktur wird darauf aufgebaut. [BF15]

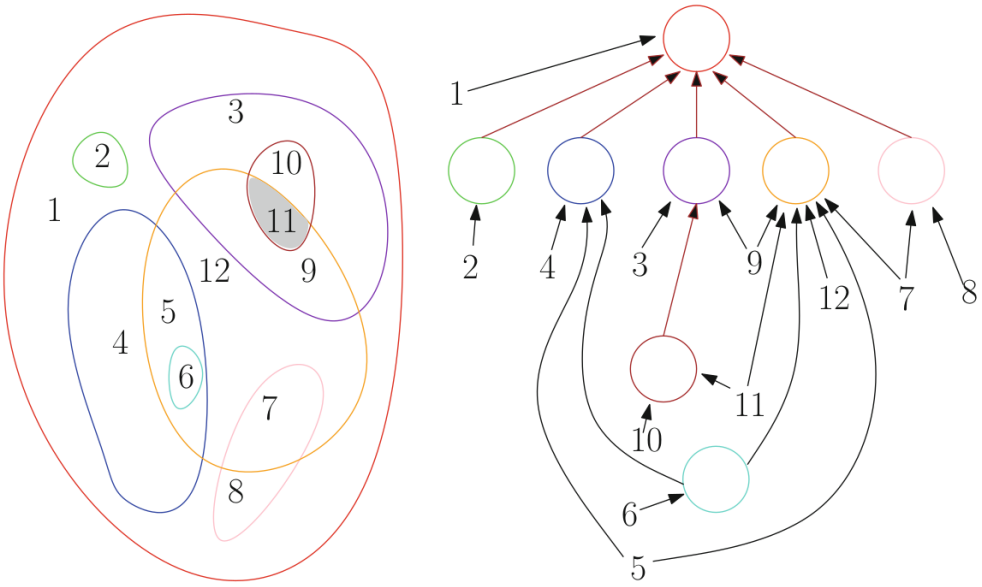

Abbildung 2.2: Links: Darstellung von Regionen. Rechts: Darstellung der dazugehörigen Zellanordnung. Quelle: [BF15] Seite 159

2.1.3 OSCAR-Architektur

OSCAR besteht aus verschiedenen Komponenten:

- oscar-create[2]
- liboscar[3]
- oscar-web[4]
- oscar-cmd[5]
- oscar-gui[6]

Oscar-create ist der Präprozessor, der OSM-Daten einliest, daraus die Zellanordnung und Suchstruktur generiert und in die Suchdaten schreibt. Das Modul ist in C++ geschrieben, läuft auf einem leistungsstarken Rechner mit viel Arbeitsspeicher (256 GiB) und benötigt ungefähr einen Tag, um die Suchdaten zu erstellen. [BF15]

[2] https://github.com/dbahrdt/oscar/tree/master/oscar-create
[3] https://github.com/dbahrdt/liboscar
[4] https://github.com/dbahrdt/oscar-web
[5] https://github.com/dbahrdt/oscar/tree/master/oscar-cmd
[6] https://github.com/dbahrdt/oscar/tree/master/oscar-gui

Oscar-web besteht aus der Webseite und der Webserverapplikation. Die Webseite stellt ihre Oberfläche mit HTML/CSS dar, sendet mittels JavaScript Anfragen an den Webserver und zeigt die Ergebnisse mithilfe der Bibliothek leaflet[7] auf einer Weltkarte an. Leaflet ist eine JavaScript-Bibliothek, mit der interaktiv geografische Kartendaten dargestellt werden können.

Die Webserver-Applikation ist in C++ geschrieben und benutzt das Framework cppcms[8], um HTTP-Schnittstellen für die Webseite anzubieten. Vom Webserver werden die angebotenen Funktionen aus liboscar benutzt, um die von oscar-create erstellten Dateien zu durchsuchen und sie dann in eine für die Webseite lesbaren Datenstruktur zu bringen. [BF15]

Diese Arbeit besteht aus einer Erweiterung der Webserverapplikation und der Weboberfläche, also von oscar-web.

Oscar-cmd und oscar-gui bieten eine Benutzerschnittstelle zu liboscar, wobei das erste ein Kommandozeileninterface und das zweite eine Benutzeroberfläche auf Basis des qt-frameworks implementiert. Beide werden in dieser Arbeit nicht weiter behandelt.

2.1.4 OSCAR-Suchsprache

Eine OSCAR-Suchanfrage kann unter anderem aus Schnitt-, Vereinigungs- und Ausschlussoperationen bestehen. Dabei ist der Schnitt durch eine Leerstelle gegeben, die Vereinigung durch ein '+' und der Ausschluss durch ein '-'. Will nach bestimmten Keys aus den OSM-Tags gesucht werden, werden diese mit '@key' referenziert. Nach Key-Value-Paaren wird analog mit '@key:value' gesucht. Soll explizit angefordert werden, dass ein Suchwort eine bestimmte Region beschreibt, wird dieses Wort mit Anführungszeichen umschlossen. Falls dies nicht durchgeführt wird, werden auch Ergebnisse angezeigt, die das Suchwort zum Beispiel im Namen oder Straßennamen haben. [BF15]

So zeigt die Suchanfrage 'hotel München' auch Ergebnisse an, die nicht in München liegen, sondern zum Beispiel auf der Münchener Straße in Ingolstadt. Um die Ergebnismenge auf München zu beschränken, muss die Suchanfrage 'hotel "München"' benutzt werden.

Zur Veranschaulichung wird in Abbildung 2.3 das Ergebnis der Suchanfrage '@cuisine:japanese "Stuttgart"- "Stuttgart-Mitte"' dargestellt. Diese bewirkt, dass alle japanischen Restaurants, die in Stuttgart aber nicht in Stuttgart-Mitte liegen, angezeigt werden.

[7] https://leafletjs.com/

[8] http://cppcms.com/wikipp/en/page/main

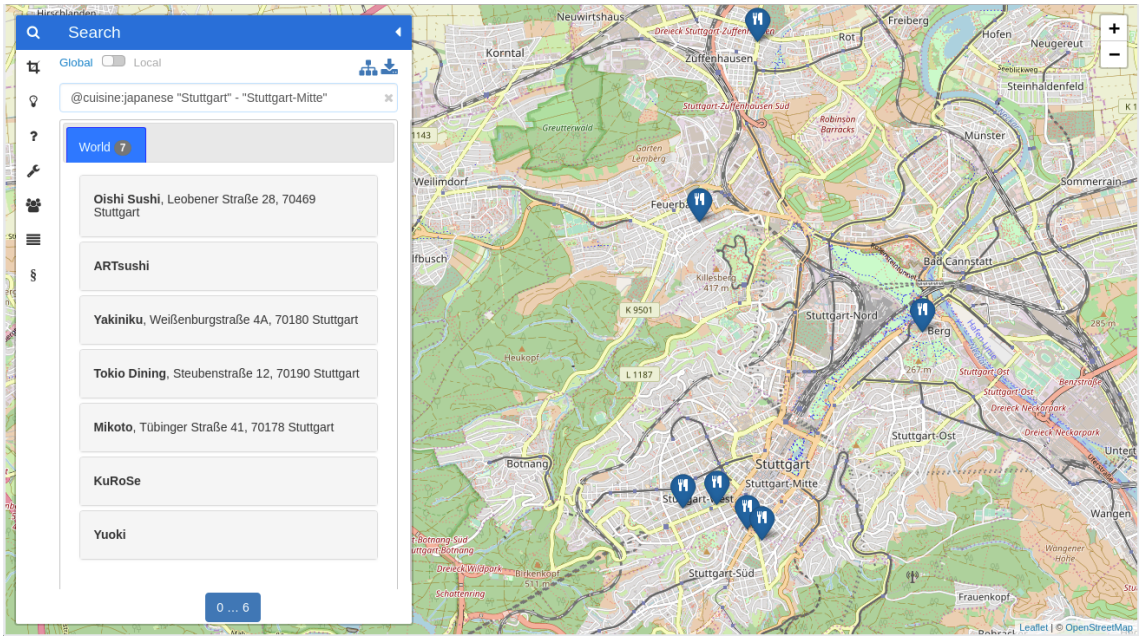

Abbildung 2.3: Beispiel Suchanfrage auf der OSCAR-Weboberfläche

2.2 Strukturierung von Kopf

Die Aufgabe dieser Arbeit ist es, die Algorithmen von Kopf [Kop17], die in Java implementiert sind, umzuschreiben in C++ und gegebenenfalls so abzuwandeln, dass sie die OSCAR-Webapplikation erweitern und effizient auf dem Webserver von OSCAR laufen. Deswegen werden in diesem Kapitel zunächst die Algorithmen von Kopf [Kop17] erläutert.

2.2.1 Regionales Clustering

Durch die von OSCAR gegebene Zellstruktur können zu einer Suchanfrage effizient alle Resultate und deren regionale Gliederung gefunden werden. Nun können allen Regionen, auch Parents genannt, die in ihr liegenden Objekte zugeordnet werden. Das Ziel ist jetzt, zu jeder Suchanfrage eine Teilmenge dieser Regionen zu finden, die besonders sinnvoll erscheint.

Nach Kopf [Kop17] besteht eine gute Teilmenge aus Regionen, die viele Objekte enthalten. Dabei sollen sich diese Mengen jedoch nicht schneiden. Das hat die Folge, dass jedes Objekt nur in einer Region des angezeigten Clusterings enthalten ist.

Zum Beispiel, wenn ein Ergebnis der Suche ein Restaurant in Stuttgart ist, wäre dieses sowohl in der Region Regierungsbezirk Stuttgart sowie in der Region Baden-Württemberg enthalten. Das Clustering würde jedoch, wegen der Forderung nach schnittfreien Mengen nur eine oder keine der beiden Regionen liefern.

Um dieses Clustering zu erzeugen, schlägt Kopf [Kop17] einen Algorithmus vor, der zunächst alle Regionen nach der Anzahl ihrer Objekte absteigend sortiert. Danach werden die zwei größten sich nicht schneidenden Regionen gesucht (siehe Algorithmus 2.1). Auf Basis der gefundenen

Algorithmus 2.1 Algorithmus, der die zwei größten Parents p_i und $p_j \in P$ findet, wobei p_i und p_j sich nicht schneiden und $n = |P|$. Quelle: [Kop17] Seite 18

```
for i = 2 to n do
    for j = 1 to i - 1 do
        if |p_i ∩ p_j| = 0 then
            add p_i, p_j to resultlist
            end algorithm
        end if
    end for
end for
```

Algorithmus 2.2 Algorithmus, der alle weiteren schnittfreien Parents in P findet, wobei sich die zwei größten Parents schon in der Ergebnismenge resultlist befinden. Quelle: [Kop17] Seite 19

```
for k do = i + 1 to n
    for all l : p_l ∈ resultlist do
        if |p_k ∩ p_l| ≠ 0 then
            discard p_k
        end if
    end for
    if p_k is not discarded then
        add p_k to resultlist
    end if
end for
```

Regionen, werden alle weiteren der Größe nach durchgegangen. Es werden nur die Regionen dem Ergebnis hinzugefügt, die andere Regionen, die bereits im Ergebnis sind, auch nicht schneiden (siehe Algorithmus 2.2).

Um die Schnittmengenbildung effizient durchzuführen, schlägt Kopf [Kop17] vor, alle Regionen und ihre Items in ein Array, das aus Region-Item-Paaren besteht, einzusetzen. Jedoch reicht es für die effiziente Implementierung in C++, jeder Region einen Vektor mit den Objekt-IDs zuzuweisen und diese Vektoren dann zu sortieren. Dadurch wird die Regions-ID auch nur einmal in den Speicher gelegt und die gesamte Datenstruktur verbraucht somit weniger Speicherplatz.

Durch die Sortierung kann mit dem Algorithmus 2.3 die Schnittmengenbildung in $O(n + m)$ durchgeführt werden. [Kop17]

Da immer nach schnittfreien Mengen gesucht wird, kann der Algorithmus beendet werden, sobald ein Schnitt gefunden wurde. Um diesen aber allgemein zu halten, wird er nach einer gegebenen Anzahl an Schnitten gestoppt (siehe Algorithmus 2.3).

Um die Strukturierung durchzuführen, muss also zunächst allen Objekten ihre zugehörigen Parents zugeordnet werden, damit danach die größten Parents ohne Schnitt gefunden werden können.

Algorithmus 2.3 Schnittmengenberechnung einer zwischen zwei sortierten Mengen.

```
//returns true if the number of intersections is greater than minNumber
template<typename It>
bool hasIntersection(It beginI, It endI, It beginJ, It endJ, const std::float_t &minNumber)
{
  std::uint32_t intersectionCount = 0;
  while (beginI != endI && beginJ != endJ) {
    if (*beginI < *beginJ) ++beginI;
    else if (*beginJ < *beginI) ++beginJ;
    else {
      //intersection found increment both
      ++beginI;
      ++beginJ;
      if (++intersectionCount > minNumber) {
        return true;
      }
    }
  }
  return false;
}
```

2.2.2 Key-Value- und Key-Clustering

Durch die Struktur von OSM können die Objekte auch nach Key-Value-Paaren oder Keys geclustert werden. So werden dann zum Beispiel Objekte dem Key-Value-Paar 'cuisine:japanese' und andere Objekte dem Paar 'cuisine:chinese' zugeordnet. Es wäre somit sinnvoll dem Benutzer eine Strukturierung nach Keys oder Key-Value-Paaren zu geben, sodass dieser die Verfeinerungsmöglichkeiten sieht und erkennt, wie viele Objekte die jeweiligen Key-Value-Paare oder Keys enthalten.

Da das Key-Value- und Key-Clustering analog durchgeführt wird, wird von jetzt an nur vom Key-Value-Clustering die Rede sein.

Kopf [Kop17] führt das Key-Value-Clustering im Prinzip genauso durch wie das Clustering nach Regionen. Es werden zunächst allen Key-Value-Paaren deren Objekte zugeordnet. Die Objekte jedes Key-Value-Paars werden nach Objekt-IDs sortiert, um, wie beim regionalen Clustering, eine schnellere Schnittmengenbildung zu erlauben.

Um zu verhindern, dass ein Key-Value-Paar alle anderen verdrängt, argumentiert Kopf [Kop17], dass ein Fordern nach schnittfreien Mengen zu stark wäre. Somit schlägt Kopf [Kop17] vor, als Schnittrestriktion $|P_i \cap P_j| \leq (|P_i| + |P_j|)/200$ zu nehmen. Dabei steht P_i beziehungsweise P_j für die jeweilige Objektmenge der Keys oder Key-Value-Paare. Diese Schnittrestriktion wird auch von der Implementierung dieser Arbeit so übernommen.

Beim Key-Value-Clustering muss, statt nur durch alle Zellen zu iterieren, durch alle Objekte iteriert werden. Dadurch ist das Sammeln der Ergebnisobjekte in der Praxis langsamer als beim regionalen Algorithmus, da dabei die Zellstrukturierung von OSCAR verwendet werden kann. Es hat sich nach Bahrdt und Funke [BF15] herausgestellt, dass die Anzahl der Zellen bei den meisten Suchanfragen viel geringer ist als die Anzahl an Objekten.

2.2.3 Overhead durch Netzwerkübertragung

Da die Strukturierung von Kopf [Kop17] auf der Clientseite durchgeführt wird, müssen zunächst die Suchergebnisse einer OSCAR-Suchanfrage über das Netzwerk übermittelt werden. Dadurch werden die Daten in eine semistrukturierte JSON-Form umgewandelt, um sie danach im Client in eine Form zu bringen, mit der die Strukturierung effizient durchgeführt werden kann. [Kop17]

Da die Implementierung, die in dieser Arbeit präsentiert wird, direkt auf den vom OSCAR-Server angebotenen Datenstrukturen arbeitet, wird die Datenübertragung über das Netzwerk umgangen und die Strukturierung kann in kürzerer Zeit durchgeführt werden.

3 SECLUS - SErverseitiges CLUstering von Suchergebnissen

Die Webseite von SECLUS ist zur Zeit der Abgabe dieser Arbeit unter der URL http://kvclustering.oscar-web.de/ zu finden.

SECLUS ist eine Client-Server-Anwendung. Das heißt, ein Client, in diesem Fall die SECLUS-Webseite, sendet Anfragen an einen zentral gelegenen Server, der die Anfragen bearbeitet und die jeweiligen Antworten, in einer für den Client lesbaren Form, zurückschickt.

In diesem Kapitel wird zuerst die Implementierung des Servers (Abschnitt 3.1) beschrieben und danach die des Clients (Abschnitt 3.2).

3.1 Serverimplementierung

Die Serverapplikation von OSCAR ist in C++ geschrieben und verwendet das Framework cppcms[8]. Es wird ein Apacheserver zum Anbieten der statischen Webseitendateien benutzt und das Apache Modul FastCGIExternalServer, um mit einer sogenannten socket-Datei die Kommunikation mit der C++-Applikation zu gewährleisten.

Jede Serveranfrage mit dem URL-Pfad '/oscar/' wird durch das Apache Modul in die socket-Datei geschrieben. Das Framework cppcms liest diese Anfrage, ruft die durch die Anfrage bestimmte Funktion auf und schreibt die von der Funktion produzierte Antwort auch in die socket-Datei. Falls eine Antwort zu einer vorherigen Anfrage auftaucht, wird diese vom Apachemodul gelesen und über das Netzwerk an den Client weitergeleitet, der die Anfrage getätigt hat.

3.1.1 Clustering-Schnittstelle

Für SECLUS wird eine eigene Server-Schnittstelle angeboten, die über den URL-Pfad 'oscar/kvclustering/get' und über die HTTP-GET Methode ausführbar ist. In der Tabelle 3.1 werden die möglichen Übergabeparameter gelistet.

HTTP-GET ist eine HTTP-Anfragemethode, die benutzt werden soll, wenn lediglich Daten vom Server abfragt werden wollen. Die auch prominente HTTP-POST Methode ist laut HTTP-Spezifikation für Anfragen vorgesehen, die permanent Daten auf dem Server ändern. Es wird vom HTTP-Protokoll jedoch nicht überprüft, ob die Methoden zu ihrem empfohlenen Zweck verwendet werden. Somit kann auch eine GET-Methode dazu verwendet werden, Daten zu verändern und eine POST-Methode, um nur Anfragen zu tätigen. Die POST-Methode hat den Vorteil, dass separat zur URL Daten an den Server gesendet werden können.

Parameter	Typ	Funktion
q	String	Suchanfragentext
type	{”kv”,”p ”, ”k”}	Clusteringtyp: ”kv”= Key-Value Clustering ”p”= Regionales Clustering ”k”= Key Clustering
queryId	Int	ID der Suchanfrage
maxRefinements	Int	maximale Anzahl an Clusteringvorschlägen, die zurückgegeben werden
exceptions	Array	Key oder Key-Value ausnahmen, die beim Clustering nicht berücksichtigt werden.
keyExceptions	Array	Key-Präfixe, die beim Key-Value Clustering nicht berücksichtigt werden
debug	bool	Bestimmt ob Debuginformationen zurückgegeben werden

Tabelle 3.1: Tabelle mit den möglichen Parametern bei einer Clusteringanfrage

Listing 3.1 Beispiel einer GET-Anfrage. Weitere Header wurden wegen der Übersicht ausgelassen.

```
GET /oscar/kvclustering/get?queryId=2&q=beispiel&type=p&maxRefinements=10 HTTP/1.1
Host: kvclustering.oscar-web.de
Connection: keep-alive
Accept: application/json, text/javascript, */*; q=0.01
```

Die GET-Methode erlaubt es Übergabeparameter nur über die URL zu senden. Diese werden mit einem ’?’ an die URL angehängt und mit ’&’ voneinander getrennt. Weitere entweder vom Server oder HTTP-Protokoll vorbestimmte Informationen über die Anfrage werden mittels sogenannten HTTP-Headern übermittelt. So kann zum Beispiel bestimmt werden, in welchem Format die Antwort erwartet wird oder ob die Verbindung auch weiterhin aufrechterhalten werden soll.

Da bei einer Suchanfrage keine Daten permanent auf dem OSCAR-Server gespeichert werden und es versucht wird der HTTP-Spezifikation treu zu bleiben, wird für das Clustering die HTTP-GET Methode verwendet. Dadurch geht jedoch die Option verloren, die Parameter in einer für Menschen übersichtlicheren und kompakteren Form zu versenden. Ein Beispiel einer GET-Anfrage wird in Listing 3.1 dargestellt.

Listing 3.2 Beispiel einer Rückgabe der Anfrage auf die URL http://kvclustering.oscar-web.de/oscar/kvclustering/get?q=stuttgart&type=kv&maxRefinements=3&debug=false&queryId=0&exceptions=["building:yes"]&keyExceptions=["natural"]


```
{
  "clustering": [
    {
      "name": "building:yes",
      "itemCount": 1061041,
      "keyId": 13922,
      "valueId": 73094691
    },
    {
      "name": "highway:track",
      "itemCount": 194515,
      "keyId": 29149,
      "valueId": 72813398
    },
    {
      "name": "building:garage",
      "itemCount": 106576,
      "keyId": 13922,
      "valueId": 69907628
    }
  ],
  "hasMore": true,
  "queryId": 0
}
```


Damit der Client die Daten einfach verarbeiten kann, liegt die Rückgabe im JSON-Format vor. JSON steht für Java-Script-Object-Notation und wird häufig für die Übertragung von Daten über das Internet verwendet. Das liegt zum einen der einfachen Lesbarkeit für Menschen und zum anderen der einfachen Verarbeitung mittels der Browser-Programmiersprache JavaScript zugrunde.

In Listing 3.2 wird eine Beispielrückgabe gezeigt. Ein clustering-array enthält die Clusteringobjekttypen, die mit der Anfrage spezifiziert wurden. Ein hasMore boolean ist wahr, falls es weitere Verfeinerungen gibt, die angefragt werden können.

3.1.2 Sammeln der Suchergebnisse

Für die Strukturierung ist es offensichtlich nötig zunächst alle Suchergebnisse zu sammeln und diese den in ihnen enthaltenen Keys oder Key-Value-Paaren zuzuordnen. Die Zuordnung der Ergebnisse zu den OSM-Regionen ist bereits durch die OSCAR-Zellstruktur gegeben, jedoch muss diese auch in eine für das Clustering geeignete Datenstruktur überführt werden.

Listing 3.3 Listing, welche Datenstrukturen für das Sammeln der Ergebnisse benutzt werden.

```
\\map used for the key-value-clustering
std::unordered_map<std::pair<uint32_t, uint32_t>, std::vector<uint32_t>> keyValueItemMap;

\\map used for the key-clustering
std::unordered_map<uint32_t , std::vector<uint32_t>> keyItemMap;

\\map used for the regional-clustering
std::unordered_map<uint32_t, std::vector<uint32_t>> parentItemMap;
```

Mapping

Als Datenstruktur wurde die in der C++-Standardbibliothek enthaltene ungeordnete Map 'std::unordered_map' verwendet. Für die jeweiligen verschiedenen Clustering-Methoden wird jeweils ein unterschiedlicher Map-Key in die Map eingesetzt.

Beim Regionalen-Clustering und Key-Clustering besitzt die Map eine positive 32-bit-Ganzzahl als Map-Key, die in C++ mit 'uint32_t' angegeben wird und der von OSCAR festgelegten Regions-ID bzw. Key-ID entspricht. Beim Key-Value-Clustering wird ein Key-Value-Paar als Map-Key verwendet. Dieses Paar besteht einerseits aus der Key-ID und andererseits aus der Value-ID. Beide IDs sind positive Ganzzahlen.

Als Map-Value bekommt jede Map einen Vektor, der die zu jedem Map-Key zugehörigen Items als Item-ID, die als Ganzzahl angegeben wird, enthält.

In Listing 3.3 werden die verschieden Maps in C++-Schreibweise angezeigt.

Iterieren durch die Ergebnismenge

Durch die von OSCAR aufgebaute Zellstruktur, die in Abschnitt 2.1.2 beschrieben wird, können alle Suchergebnisse durchgegangen werden, indem durch alle für die Suchanfrage relevante Zellen iteriert wird.

Für diese Funktion wird von OSCAR ein Iterator zur Verfügung gestellt, der nur relevante Zellen referenziert. So kann dieser Iterator benutzt werden, um alle Keys beziehungsweise Key-Value-Paare und deren Items in die jeweilige Map zu setzen. Ein Beispiel-Code-Abschnitt, für das Einsetzen der Ergebnisse in die Map, ist in Algorithmus 3.1 gegeben.

Kopf [Kop17] iteriert beim regionalen Clustering auch durch alle Objekte, die Ergebnis der Suchanfrage sind. Jedoch bietet es sich an, durch die direkte Implementierung auf dem Server, alle Items einer Region dadurch zuzuordnen, indem alle Zellen in dieser Region vereinigt werden. Dies führt zu einem wesentlich schnelleren Algorithmus (siehe Abschnitt 4.2.2).

Algorithmus 3.1 Sammeln der Ergebnisse für die Key-Value- und Key-Strukturierung.

```
//iterate over all query result items
//it is the iterator for the result cells
for (it; it != end; ++it) {
  //iterate over all items in the cell
  for (const uint32_t& x : it.idx()) {
    //m_store.kvBaseItem(x) returns the tags for given itemPosition x
    const auto& item = m_store.kvBaseItem(x);
    //iterate over all keys-value pairs in the item
    for (uint32_t i = 0; i < item.size(); ++i) {
      //add key-value Pair or just the key and item to key to keyItemMap
      insertKey(keyItemMap, item, i);
    }
  }
}
```

3.1.3 Strukturierung der Suchergebnisse

Da es laut Kopf [Kop17] besonders sinnvoll ist nach Keys, Key-Value-Paaren und Regionen zu suchen, die in vielen OSM-Objekten enthalten sind, werden die Maps zunächst nach der Größe der Vektoren sortiert.

Die Vektoren, die die Items beinhalten, werden auch sortiert, weil dadurch die Schnittmengenberechnung mit dem Algorithmus 2.3 in linearer Zeit durchführt werden kann.

Die weitere Strukturierung erfolgt in allen Fällen wie in den Algorithmen 2.1 und 2.2 beschrieben.

Ausnahmen

Damit der Benutzer wie in [Kop17] beschrieben die Ergebnisfindung interaktiv durchführen kann, können der Clusteringanfrage Key- oder Key-Value-Ausnahmen hinzugefügt werden. Diese bestehen aus dem Key-Namen oder im Fall des Key-Value-Clusterings aus dem Key-Namen und Value-Namen getrennt mit einem Doppelpunkt und werden in einer Liste über den Parameter 'exceptions' übergeben.

Beim Sammeln der Keys beziehungsweise Key-Value-Paaren werden nämlich nur diese berücksichtigt, die nicht in der Ausnahmenmenge enthalten sind. Somit müssen immer die gerade betrachteten Keys beziehungsweise Key-Value-Paare mit den Ausnahmen verglichen werden und nur dann hinzufügt werden, wenn sie nicht in der Ausnahmenmenge aufzufinden sind.

Präfix-Ausnahmen

Es hat sich herausgestellt, dass bei der Strukturierung öfters Keys angezeigt werden, die kaum sinnvoll für einen Benutzer sind. Beispiele dafür werden in Kapitel 4 beschrieben.

Um gleichzeitig mehrere Keys auszuschließen, wird die Option gegeben der Suchanfrage eine Liste mit Präfixen zu übergeben. Sobald ein Key auftaucht, der mit einem Präfix anfängt, der in der Liste vorkommt, wird dieser Key nicht in der Key-Value-Strukturierung berücksichtigt. Dafür wird aus jedem Präfix ein Zahlenintervall berechnet und wenn eine Key-ID in diesem Intervall liegt, wird der Key nicht in die Map hinzugefügt.

Anzahl der Verfeinerungen

Kopf [Kop17] schlägt vor die Verfeinerungsanzahl dadurch zu beschränken, dass nur Verfeinerungen angezeigt werden, die mehr als 1/100 der Gesamtergebnismenge betreffen. Für eine Anzeige in einer UI erscheint es jedoch sinnvoller, immer eine fixe Anzahl an Verfeinerungen anzuzeigen und gegebenenfalls dem Benutzer die Möglichkeit geben mehr Verfeinerungen zu laden. Somit werden, falls vorhanden, so viele Verfeinerungen zurückgegeben wie in der Variablen 'maxRefinements' angegeben werden. Der Benutzer wird außerdem mit dem Rückgabewert 'hasMore' darüber informiert, ob noch weitere Verfeinerungen geladen werden können.

3.2 Implementierung der Weboberfläche

Ein weiterer Teil der Arbeit besteht aus einer Anbindung der SECLUS-Schnittstelle des Webservers mit der OSCAR-Weboberfläche. In diesem Abschnitt wird beschrieben, wie das umgesetzt wurde.

3.2.1 Grundlagen HTML, CSS und JavaScript

So wie jede Webseite, die auf modernen Browsern dargestellt werden soll, wird auch die OSCAR-Webseite mit HTML beschrieben. HTML ist eine XML-ähnliche Sprache und wurde entwickelt, um Dokumente zu strukturieren und ansehnlicher zu gestalten.

Der Inhalt wird mit sogenannten Tags umschlossen. So ist <h1> zum Beispiel der Tag für eine Überschrift und <p> der Tag für einen Paragrafen.

Die HTML-Tags können mit Attributen erweitert werden. Zum Beispiel kann mit dem Attribut id jedem Element eine ID zugewiesen werden.

Mehr Möglichkeiten um das Aussehen einer HTML-Seite zu gestalten bietet CSS. Ein sogenanntes CSS-Script kann in eine HTML-Seite angebunden werden. Dadurch können in CSS die jeweiligen HTML-Elemente unter anderem mit einer ID angesprochen und es kann zum Beispiel die Hintergrundfarbe des jeweiligen Elementes geändert werden. Da in HTML IDs eindeutig sind, bieten HTML und CSS mit dem class-Attribut die Möglichkeit mehrere Elemente mit dem gleichen CSS-Code anzusprechen.

Mit HTML und CSS können Webseiten nur statisch beschrieben werden. Häufig wird deswegen für browser-basierte Applikationen JavaScript als Programmiersprache benutzt. JavaScript ermöglicht es, HTML-Elemente dynamisch, auf Nutzereingaben oder Antworten von Serveranfragen, anzusprechen und zu ändern. In Listing 3.4 wird eine einfache HTML-Seite mit JavaScript und CSS Inhalten

Listing 3.4 Beispiel einer einfachen HTML-Seite

```
<html>
<head>
<title>Seitentitel</title>
<style>
#myHeading : {background-color: yellow}
</style>
<script>
function changeHeading(){
  document.getElementById("myHeading").innerHTML = "Ueberschrift geaendert";
}
</script>
</head>
<body>
<h1 id="myHeading">Dies ist eine Ueberschrift</h1>
<p>Dies ist ein Paragraf</p>
<button onclick="changeHeading()">Taste</button>
</body>
</html>
```

dargestellt. Der JavaScript-Code wird mit dem Tag '<script>' angegeben und der CSS-Code mit dem Tag '<style>'. In dem gezeigten Beispiel wird beim Drücken eines Buttons die Überschrift geändert.

Die OSCAR-Webseite benutzt mehrere JavaScript-Bibliotheken, um das Arbeiten mit JavaScript zu vereinfachen und um die Funktionen von JavaScript zu erweitern. Eine von OSCAR häufig benutzte Bibliothek ist JQuery[1]. JQuery bietet unter anderem Methoden an, um HTML-Elemente besser anzusprechen und um die Kommunikation mit dem Server einfacher zu gestalten.

Bootstrap[2], eine weitere von OSCAR verwendete Bibliothek, bietet Komponenten wie Buttons oder Seitenleisten an, die mit CSS und JavaScript gestaltet wurden. Und einem somit die aufwendige Gestaltung von häufig verwendeten Elementen erspart.

3.2.2 Äußerlicher Aufbau

In Abbildung 3.1 wird der Aufbau mit allen in diesem Abschnitt beschriebenen Elementen dargestellt.

Die OSCAR-Webseite besteht primär aus einer Weltkarte in der rein- und raus-gezoomt werden kann und in der primär die Suchergebnisse dargestellt werden.

Auf der linken Seite befindet sich eine Seitenleiste, die wiederum links verschiedene Tasten anbietet, mit denen der Inhalt der Seitenleiste verändert werden kann. Oben auf der Seitenleiste wird immer das Suchfeld angezeigt, dessen Eingabe den Suchtext für eine OSCAR-Anfrage bestimmt.

[1] https://jquery.com/
[2] https://getbootstrap.com/

Für SECLUS wurde eine Taste der Seitenleiste hinzugefügt, die bei Betätigung die Seitenleiste so verändert, dass oben ein Menü angezeigt wird, mit dem der Benutzer zwischen den drei Arten des Clusterings auswählen kann.

Nun kann der Benutzer eine Suchanfrage eingeben und das Clustering-Menü auswählen. Daraufhin werden ihm, abhängig von dem Clustering-Typ, eine Liste von Keys, Key-Value-Paaren oder Regionen angezeigt, mit denen er das Suchergebnis verfeinern kann.

Er kann zum einen mit einer '+'-Taste das Suchergebnis nur mit Resultaten, die diese Verfeinerung enthalten, einschränken oder zum anderen mit einer '-'-Taste diese Suchresultate ausschließen. Beim Key- und Key-Value-Clustering wird zusätzlich pro Verfeinerung die Möglichkeit angeboten die Verfeinerung auszuschließen und somit den Clustering-Algorithmus nochmal durchzuführen, jedoch ohne, dass diese Verfeinerung berücksichtigt wird.

Ganz rechts neben jeder Verfeinerung wird die Anzahl der Items angezeigt, die Keys beziehungsweise Key-Value-Paare enthalten oder sich in der Region befinden. Unter der Verfeinerungsliste befindet sich, falls mehr Verfeinerungen vorhanden sind, eine Taste die diese Lädt.

Über dem Verfeinerungsmenü werden die aktiven Verfeinerungen angezeigt. Die einschließenden Verfeinerungen sind dabei grün und die ausschließenden rot. Es kann entweder jede Verfeinerung einzeln mit einem 'x', welches neben der Verfeinerung angezeigt wird entfernt werden oder alle auf einmal mit der Betätigung der 'x'-Taste ganz rechts in der Seitenleiste.

Unter dem Verfeinerungsmenü werden dem Benutzer die aktiven Key- bzw. Key-Value-Ausnahmen angezeigt. Hier kann dieser wiederum, analog zu den aktiven Verfeinerungen, entweder jede Ausnahme einzeln entfernen oder alle auf ein Mal.

Falls der Benutzer die Zahnräder oben links auf der Seitenleiste betätigt, kann er die Präfixausnahmen, die in Abschnitt 3.1.3 beschrieben sind, bearbeiten. Dabei öffnet sich ein Fenster, das zum einen ein Eingabefeld zeigt, in dem die Präfixliste eingegeben werden kann, und zum anderen zwei Tasten, mit denen entweder die Eingabe gespeichert oder die Präfixliste auf die Voreinstellung zurückgesetzt wird. Die Präfixe sind jeweils mit Anführungszeichen umschlossen und mit Kommata getrennt. Die visuelle Darstellung wird in Abbildung 3.2 angezeigt.

3.2.3 Implementierung in HTML und CSS

Grundlegend für die HTML und CSS Implementierung ist, die von Twitter veröffentlichte Bibliothek Bootstrap[2]. Diese bietet unter anderem verschiedene vorgestylte HTML-Komponenten an, sodass ohne viel Aufwand Strukturen wie Sidebars oder Menüs erstellt werden können.

Bootstrap besteht vor allem aus CSS-Styles die mit HTML-Klassen angesprochen werden. So kann zum Beispiel, um den von Bootstrap definierten Button-Style zu benutzen, einem HTML-Button die Klasse 'btn' zugewiesen werden.

OSCAR benutzt eine Bootstrap-Sidebar. In HTML ist das ein Element, das die Klasse 'sidebar' hat. Für das Clustering wird in der Sidebar ein eigenes Element hinzugefügt. Darin befindet sich ein Bootstrap-Menu mit drei Tabs, für jede Clustering-Methode eins.

Die aktiven, die vorgeschlagenen und die Verfeinerungen in den Ausnahmen sind jeweils HTML-Listen.

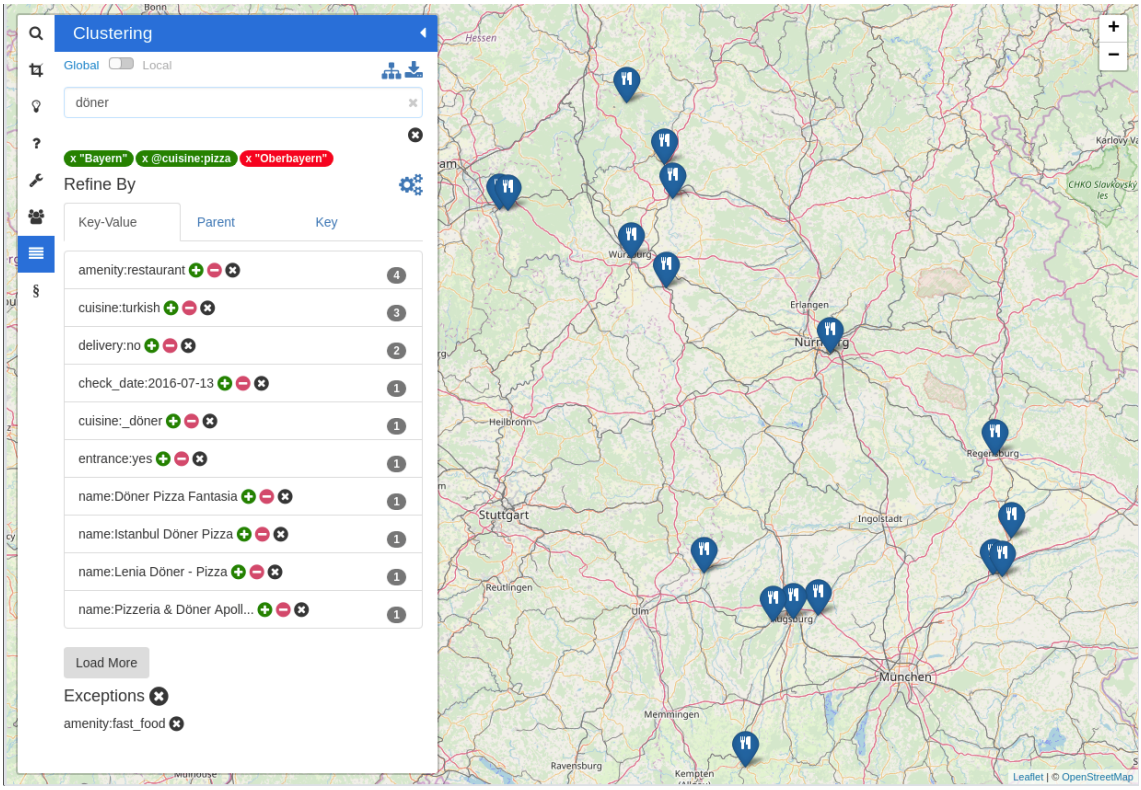

Abbildung 3.1: Beispiel der Clustering-Oberfläche mit aktiven Verfeinerungen, aktiven Ausnahmen und einer Verfeinerungsliste mit Key-Value Verfeinerungen

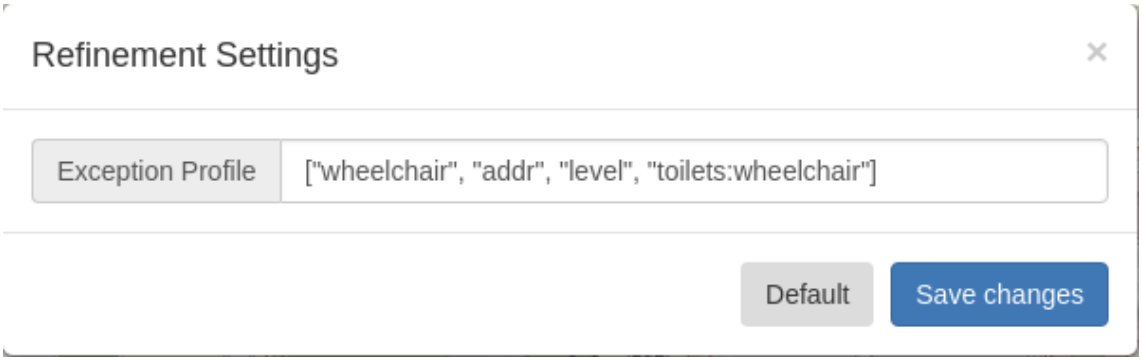

Abbildung 3.2: Darstellung der Präfixausnahmen.

Solange eine vorgeschlagene Verfeinerung noch geladen wird, wird anstelle der Verfeinerung ein pulsierender grauer Balken angezeigt. Dies hat den Vorteil, dass der Benutzer zum einen weiß, dass gerade ein Ladevorgang stattfindet und zum anderen an welcher Stelle die Daten dann angezeigt werden. Würde andererseits nur ein Ladebalken für alle Daten in der Liste benutzt werden, der nicht genauso groß ist wie die Liste nach dem Laden, werden alle Elemente unter der Liste verschoben, sobald die Ergebnisse da sind. Wie die Liste beim Laden aussieht, wird in Abbildung 3.3 dargestellt.

Abbildung 3.3: Anzeige der Verfeinerungen während dem Laden

Das Exceptionfenster (siehe Abbildung 3.2) ist ein sogenanntes Bootstrap-Modal. Dieses wird durch eine Buttonbetätigung geöffnet.

3.2.4 JavaScript-Implementierung

Grundlegend für die Implementierung der Darstellungslogik ist ein JavaSript-Objekt das Clustering-State genannt wird. Alles was angezeigt wird, ist im State enthalten. Sobald sich der State ändert, wird die Anzeige neu gezeichnet. Stateänderungen ergeben sich entweder durch Nutzereingaben oder durch geladene Daten aus dem OSCAR-Server. Der State im Initialzustand wird in Listing 3.5 angezeigt.

Wird für eine Suchanfrage vom Benutzer der Clustering-Tab ausgewählt, so wird mittels JavaScript eine Anfrage an die Clustering-Schnittstelle des Oscar-Servers gesendet. Solang die Antwort des Servers noch nicht angekommen ist, werden die Ladebalken aus Abbildung 3.3 angezeigt. Wenn die

Listing 3.5 Initialzustand der Weboberfläche

```
{
openedClustering : '#p-tab',
kvQueryID : -1,
kQueryID : -1,
pQueryID : -1,
kRefinements :
  tools.SimpleHash(), // keyID -> {name : String, itemCount: int}
pRefinements :
  tools.SimpleHash(), // parentID -> {name : String, itemCount: int}
kvRefinements :
  tools.SimpleHash(), // "{keyId: int, valueId: int}" -> {name: String, itemCount: int}
activeIncludingRefinements: [],
activeExcludingRefinements: [],
kvExceptions:
  tools.SimpleHash(), // "{keyId: int, valueId: int}" -> {name : String, itemCount: int}
kExceptions:
  tools.SimpleHash(), // keyID -> {name: String, itemCount: int}
lastKvQuery: "",
lastKQuery: "",
lastPQuery: "",
lastQueryWithoutRefinements: "",
kvRefinementCount: 10,
kRefinementCount: 10,
pRefinementCount: 10,
kvHasMore: false,
kHasMore: false,
pHasMore: false,
exceptionProfile:
  '["wheelchair", "addr", "level", "toilets:wheelchair", "building", "source"]',
}
```

Antwort empfangen wurde, wird zunächst mittels der queryID entschieden, ob diese die Antwort auf die zuletzt getätigte Anfrage ist. Ist dies nicht der Fall, wird die Serverantwort nicht weiter betrachtet. Das hat den Vorteil, dass die Anzeige nicht von Daten, die das Ergebnis von früher getätigten Anfragen sind, überschrieben wird.

Wird die Antwort akzeptiert, werden die Daten in den entsprechenden State-Eintrag geschrieben und die Anzeige wird aktualisiert.

Für jede Anzeige in der Oberfläche wird eine „draw“-Funktion implementiert. Diese verändert die HTML-Darstellung dadurch, dass sie die im State beschriebenen Objekte in eine HTML-Schreibweise umwandelt. Das geschieht, indem mit JavaScript das benötigte Element angesprochen und verändert wird. Wenn es sich jedoch um eine Liste handelt, werden Elemente in die HTML-Liste einfügt.

Wird eine Verfeinerung den aktiven Verfeinerungen hinzugefügt, so sendet JavaScript eine Query-Anfrage an den OSCAR-Server, sodass nur noch die Elemente auf der Karte angezeigt werden, die der Verfeinerung entsprechen.

4 Experimente

In diesem Kapitel werden einige Exprimente durchgeführt, die zum einen die Korrektheit der Strukturierungsergebnisse (Abschnitt 4.1) und zum anderen die Geschwindigkeit, mit der der SECLUS-Server die Ergebnisse berechnet, zeigen sollen.

4.1 Beispielhafte Ergebnisse

4.1.1 Suche nach Hotels

Als erstes Beispiel wird die Suchanfrage „hotel München“ aus Abschnitt 2.1.4 gewählt. Wenn diese nun betätigt und das regionale Clustering ausgesucht wird, werden die verschiedenen Regionen und die Anzahl der Ergebnisse für jede Region angezeigt (siehe Abbildung 4.1). Der Benutzer kann dann das grüne Plus drücken und es werden tatsächlich nur Ergebnisse in München angezeigt, falls München als Verfeinerung ausgewählt wurde.

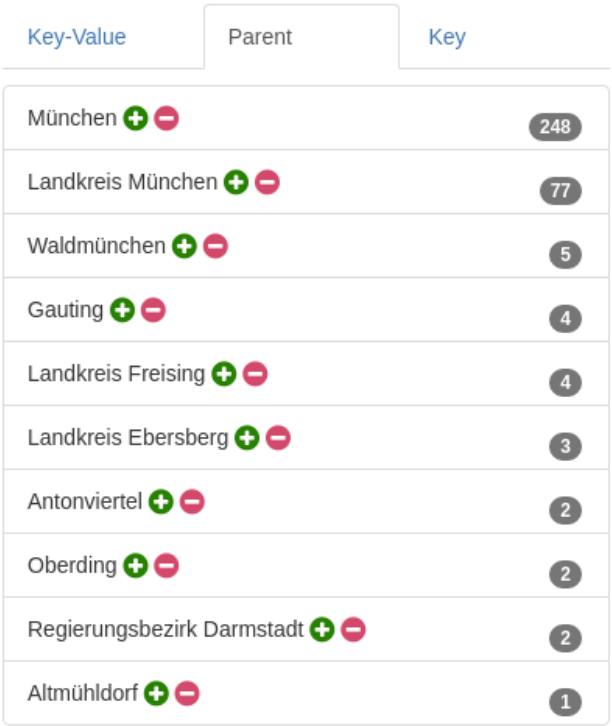

Abbildung 4.1: Anzeige des Regionalen Clusterings mit der Suchanfrage ’hotel München’.

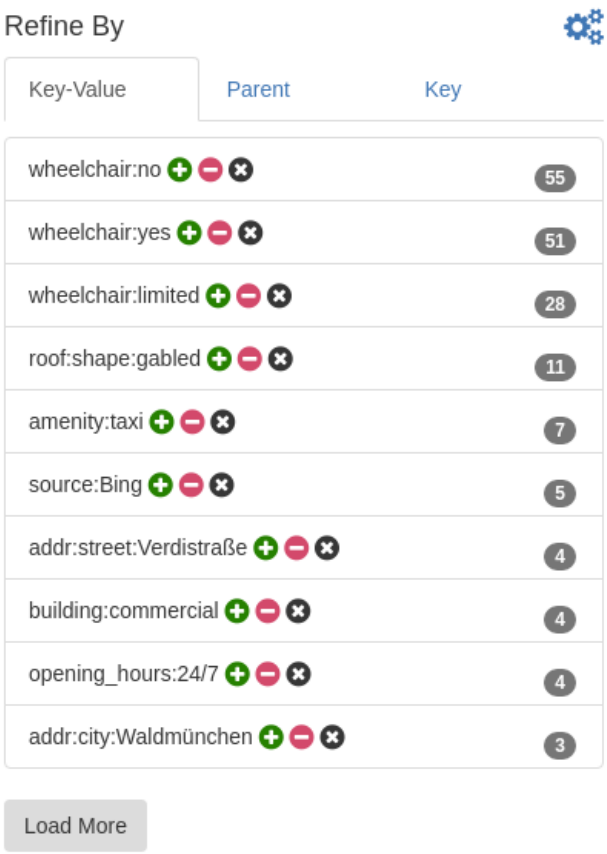

Abbildung 4.2: Anzeige des Key-Value-Clusterings mit der Suchanfrage 'hotel München'.

Das Key-Value-Clustering ohne Präfix-Ausnahmen wird in Abbildung 4.2 angezeigt. Man sieht, dass die am meisten vorkommenden Key-Value-Paare „wheelchair:no“, „wheelchair:yes“ und „wheelchair:limited“ sind. Wenn diese Paare nicht interessant sind, können sie einzeln mit einem Klick auf das schwarze X entfernt werden. Danach wird das in Abbildung 4.3 dargestellte Clustering geliefert. Dieses Clustering ist für den durchschnittlichen Benutzer vorteilhafter, denn es werden Verfeinerungen wie zum Beispiel die Anzahl der Sterne, die das Hotel hat, angeboten.

Es stellt sich heraus, dass einige Key-Value-Paare für den Benutzer uninteressant sind. Zum Beispiel diejenigen, die mit „addr“ beginnen. Somit werden standardmäßig die Präfixe „wheelchair“, „addr“, „level“, „toilets:wheelchair“, „building“, „roof“ und „source“ ausgeschlossen.

Mit diesem Ausnahmenprofil werden die in Abbildung 4.4 dargestellten Verfeinerungen berechnet. Man sieht, dass mehr Key-Value-Paare mit der Anzahl an Sternen vorgeschlagen werden und auch ob das Hotel eigene Parkmöglichkeiten anbietet. Wenn nun noch die Namen ausgeschlossen werden, werden auch andere Sternenanzahlen angezeigt.

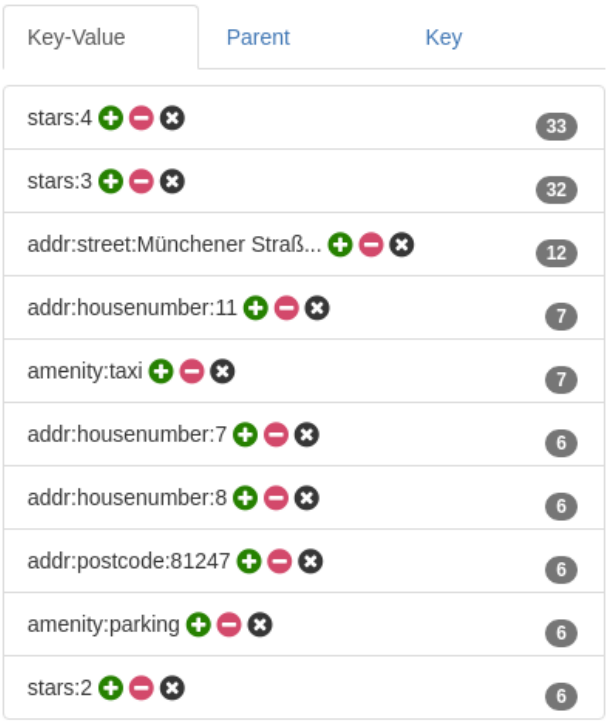

Abbildung 4.3: Anzeige des Key-Value-Clusterings mit der Suchanfrage ’hotel München’, nachdem das Paar „wheelchair:yes“ entfernt wurde.

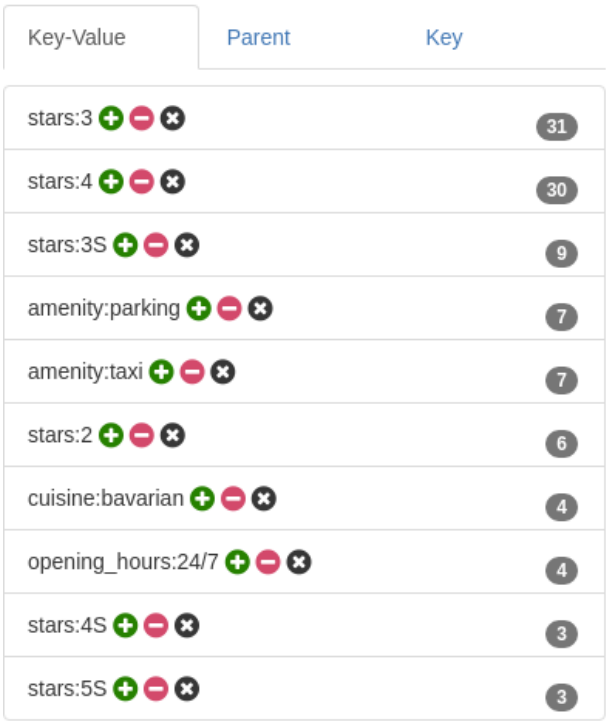

Abbildung 4.4: Anzeige des Key-Value-Clusterings mit der Suchanfrage ’hotel München’ mit aktivierten Standardausnahmenprofil.

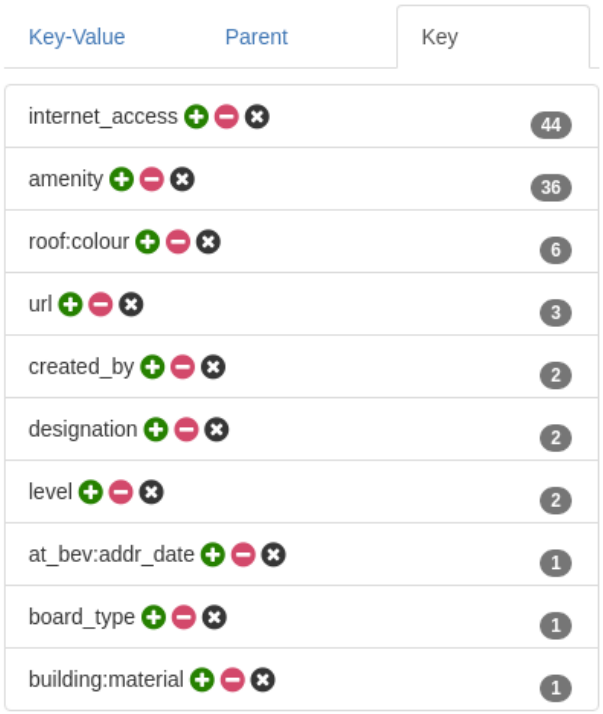

Abbildung 4.5: Anzeige der Key-Verfeinerungen mit der Suchanfrage 'hotel München'.

Der Vollständigkeit halber wird in Abbildung 4.5 das Ergebnis der Key-Strukturierung gezeigt. Diese liefert aber, auch nach dem Ausschließen von Keys, keine relevanten oder sogar irreführende Verfeinerungen. Denn wenn zum Beispiel die Verfeinerung „internet_access“ gewählt wird, könnte fälschlicherweise vermutet werden, dass nur Hotels dargestellt sind, die eine Anbindung zum Internet bereitstellen. Jedoch könnte der Wert des Keys auch „no“ sein. Und somit werden auch Hotels angezeigt, die keine Internet-Verbindung anbieten.

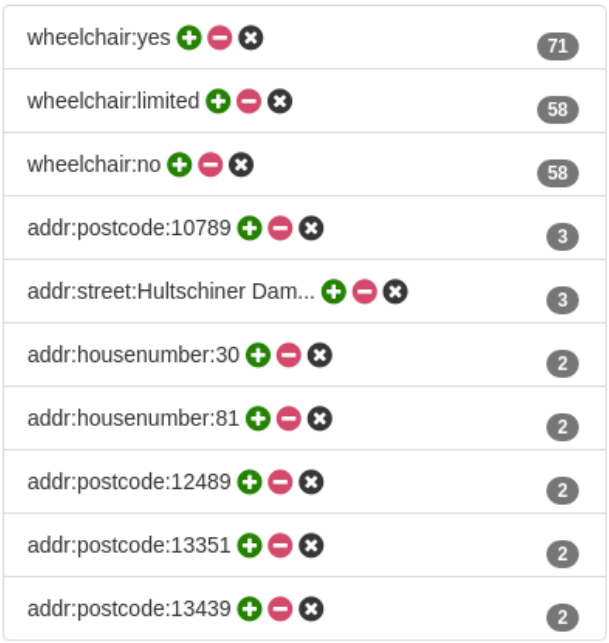

Abbildung 4.6: Key-Value-Strukturierung der Ergebnisse der Suchanfrage „restaurant ”Berlin”“.

4.1.2 Suche nach Restaurants

Als nächstes wird die Suche nach Restaurants betrachtet. Um eine realistische Suchanfrage zu simulieren, sollen nur Ergebnisse in Berlin angezeigt werden. Dieses kann mit der Suchanfrage „restaurant ”Berlin”“ erreicht werden. Die Key-Value-Strukturierung ohne Ausnahmen wird in Abbildung 4.6 dargestellt.

Die „addr“ Verfeinerungen erscheinen wenig sinnvoll und auch die „wheelchair“ Verfeinerungen sind nur für Menschen mit Rollstuhl wichtig. Beim Anwenden der Standardausnahmen ergibt sich jedoch ein ganz anderes Bild (siehe Abbildung 4.7). Es werden die Restaurants nach Essensarten unterschieden. So werden zum Beispiel Verfeinerungen nach chinesischem, regionalem oder indischem Essen angeboten.

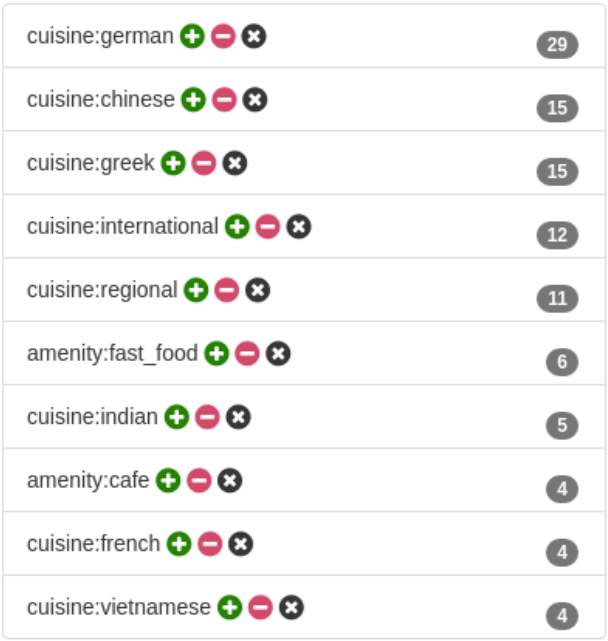

Abbildung 4.7: Key-Value-Strukturierung der Ergebnisse der Suchanfrage „restaurant ”Berlin”“ mit Ausnahmenprofil.

4.1.3 Suche nach Supermärkten

Bei der Suche nach Supermärkten sieht man zum einen in Abbildung 4.8, dass die Strukturierung ohne Ausnahmen wieder Ergebnisse liefert, die für die meisten Benutzer nicht nützlich sind. Zum anderen sieht man jedoch in Abbildung 4.9, dass die Supermärkte mit dem Standardausnahmenprofil nach Namen strukturiert werden. Dies scheint die beste Art zu sein, Supermärkte zu strukturieren.

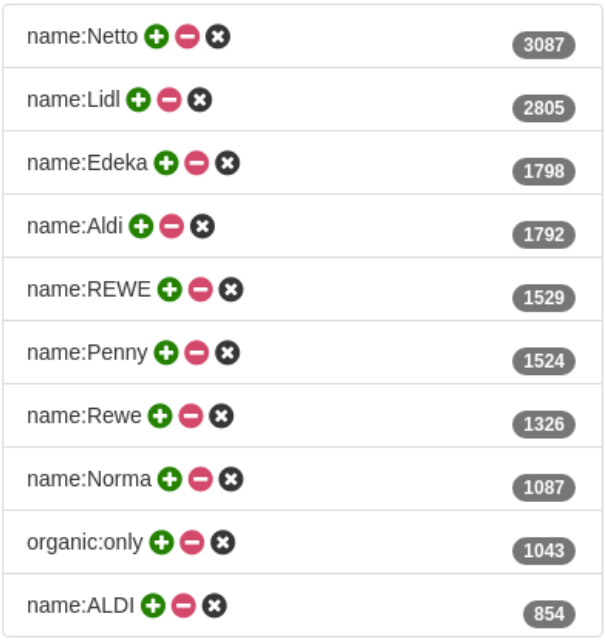

Abbildung 4.9: Key-Value-Strukturierung der Ergebnisse der Suchanfrage „@amenity:supermarket ”Deutschland”“ mit Ausnahmenprofil.

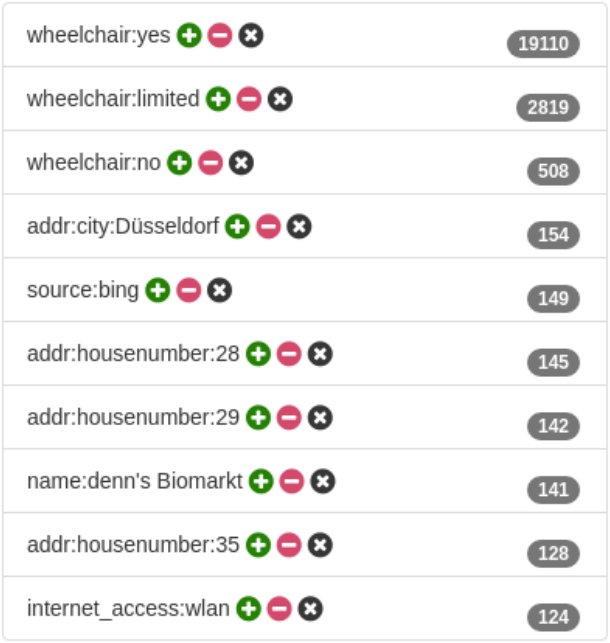

Abbildung 4.8: Key-Value-Strukturierung der Ergebnisse der Suchanfrage „@amenity:supermarket ”Deutschland”“ ohne Ausnahmenprofil.

Suchanfrage (Ergebnisanzahl)	Dauer Clientclustering	Dauer SECLUS
hotel (15698)	992 ms	76 ms
@highway ”Bremen” (63759)	1632 ms	178 ms
Landkreis Heidenheim (106307)	2452 ms	83 ms
@highway ”Berlin” (163005)	3904 ms	159 ms
Landkreis Böblingen (206608)	5420 ms	201 ms
@highway:secondary (365660)	13162 ms	669 ms
@highway ”Rheinland-Pfalz” (639241)	19953 ms	635 ms
ba ”Sachsen” (893091)	23780 ms	716 ms
straßenbahn (934873)	27977 ms	826 ms

Tabelle 4.1: Tabelle, die die Geschwindigkeit des Key-Value-Clusterings auf dem Client mit der des Clusterings auf dem Server vergleicht.

4.2 Benchmarks

In diesem Kapitel wird das serverseitige Clustering SECLUS, das Thema dieser Arbeit ist, mit dem clientseitigen Clustering, das Thema von Kopf [Kop17] ist, verglichen. Da wegen der Zellstruktur, die Oscar aufbaut (auch besprochen in Abschnitt 3.1.2), die regionale Strukturierung deutlich schneller ist als die Key- oder Key-Value-Strukturierung, werden diese separat betrachtet.

Die Algorithmen wurden auf einem „Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz“ ausgeführt und es wurde der Kartendatensatz von Deutschland verwendet.

4.2.1 Key-Value und Key-Benchmarks

Wie in Tabelle 4.1 gesehen werden kann, ist das Key- und Key-Value-Clustering von SECLUS um einiges schneller als das Clustering auf der Clientseite.

Der Geschwindigkeitsunterschied lässt sich damit begründen, dass für die Strukturierung auf Clientseite die Daten zunächst in ein Format umgewandelt werden müssen, das übers Internet versendet werden kann. Danach müssen die Daten vom Client komplett gelesen und wieder in ein für die Algorithmen effizientes Format umgewandelt werden.

Im Gegensatz dazu liegen, wenn die Strukturierung direkt auf dem Server durchführt wird, die Daten schon in einer effizienten Struktur vor und dieser Kommunikationsoverhead entsteht nicht.

Man kann eine Verbesserung der Geschwindigkeit um das Zehn- bis Zwanzigfache beobachten.

Es stellt sich jedoch auch heraus, dass selbst ohne den Kommunikationsoverhead der Serveralgorithmus schneller läuft (siehe Tabelle 4.2). Das lässt sich zum einen über eine optimierte Implementierung begründen und zum anderen dadurch, dass der Code von Kopf [Kop17] in Java geschrieben ist und der Server-Code von SECLUS in C++. Kompilierter C++-Code wird direkt auf Maschinenebene ausgeführt. Kompilierter Java-Code muss hingegen von einer virtuellen Maschine zur Laufzeit interpretiert werden und ist somit langsamer als C++.

Suchanfrage (Ergebnisanzahl)	Dauer Clientclustering	Dauer SECLUS
hotel (15698)	279 ms	76 ms
@highway ”Bremen” (63759)	429 ms	178 ms
Landkreis Heidenheim (106307)	585 ms	83 ms
@highway ”Berlin” (163005)	904 ms	159 ms
Landkreis Böblingen (206608)	1578 ms	201 ms
@highway:secondary (365660)	2481 ms	669 ms
@highway ”Rheinland-Pfalz” (639241)	2164 ms	635 ms
ba ”Sachsen” (893091)	3485 ms	716 ms
straßenbahn (934873)	2832 ms	826 ms

Tabelle 4.2: Tabelle, die die Geschwindigkeit des Key-Value-Clusterings auf dem Client mit der des Clusterings auf dem Server vergleicht. Wobei beim Clientclustering der Kommunikationsoverhead abgezogen wurde.

Suchanfrage (Ergebnisanzahl)	Dauer Clientclustering	Dauer SECLUS
hotel (15698)	842 ms	19 ms
@highway ”Bremen” (63759)	2955 ms	26 ms
Landkreis Heidenheim (106307)	2441 ms	10 ms
@highway ”Berlin” (163005)	5446 ms	14 ms
Landkreis Böblingen (206608)	4919 ms	10 ms
@highway:secondary (365660)	14307 ms	95 ms
@highway ”Rheinland-Pfalz” (639241)	16539 ms	28 ms
ba ”Sachsen” (893091)	24070 ms	24 ms
straßenbahn (934873)	28918 ms	64 ms

Tabelle 4.3: Tabelle, die die Geschwindigkeit des Regionalen-Clusterings auf dem Client mit der des Clusterings auf dem Server vergleicht.

4.2.2 Benchmarks des Regionalen Clusterings

Die Ausnutzung der Zellstruktur von OSCAR führt zu dem Ergebnis, dass die regionale Strukturierung um einiges schneller ist als die Implementierung von Kopf [Kop17]. Dies sieht man zum einen an dem Vergleich in der Tabelle 4.3.

Deutlicher zeigen sich jedoch die Unterschiede, wenn man die Implementierungen ohne Kommunikationsoverhead zwischen Client und Server anschaut (siehe Tabelle 4.4).

Die Berechnungen finden selbst bei großen Datenmengen in unter 100 ms statt. Somit ist SECLUS teilweise um das Hundertfache schneller als die clientseitige Implementierung.

Suchanfrage (Ergebnisanzahl)	Dauer Clientclustering	Dauer SECLUS
hotel (15698)	103 ms	19 ms
@highway ”Bremen” (63759)	361 ms	26 ms
Landkreis Heidenheim (106307)	467 ms	10 ms
@highway ”Berlin” (163005)	470 ms	14 ms
Landkreis Böblingen (206608)	814 ms	10 ms
@highway:secondary (365660)	1102 ms	95 ms
@highway ”Rheinland-Pfalz” (639241)	1504 ms	28 ms
ba ”Sachsen” (893091)	1812 ms	24 ms
straßenbahn (934873)	2142 ms	64 ms

Tabelle 4.4: Tabelle, die die Geschwindigkeit des Regionalen-Clusterings auf dem Client mit der des Clusterings auf dem Server vergleicht. Wobei beim Clientclustering der Kommunikationsoverhead abgezogen wurde.

5 Zusammenfassung und Ausblick

In dieser Arbeit wurde zuerst die Suchmaschine auf geografischen Daten OSCAR vorgestellt. Danach wurden die Algorithmen von Kopf [Kop17] analysiert, die die Suchergebnisse von OSCAR auf Clientseite in eine bessere Struktur bringen, um dann eine Serverimplementierung der Strukturierungsfunktionalität, namens SECLUS, zu präsentieren und im Detail zu erklären. Es wurden unterschiedliche Beispiele aufgeführt, die die Nützlichkeit der Strukturierungsergebnisse von SECLUS aufzeigen. Durch einen Vergleich zwischen Kopf [Kop17] und SECLUS wurde außerdem gezeigt, dass SECLUS um das Zehn- bis Zwanzigfache und teilweise auch um das Hundertfache schneller ist als die Strukturierung von Kopf [Kop17].

Ausblick

Die Geschwindigkeit der Serverimplementierung könnte damit weiter verbessert werden, indem das Sammeln der Suchergebnisse Gebrauch von heutigen Mehrkernprozessoren macht. Dafür müssen die Algorithmen so verändert werden, dass das Sammeln nicht sequentiell ausgeführt wird, sondern nebenläufig.

Außerdem wäre eine stärkere Kopplung zwischen der normalen OSCAR-Oberfläche und der SECLUS-Oberfläche von Vorteil. Es kann damit dem Benutzer immer eine Anzahl an Verfeinerungen angeboten werden, obwohl dieser das SECLUS-Feld gar nicht explizit ausgewählt hat.

Alle URLs wurden zuletzt am 12. 11. 2018 geprüft.

Literaturverzeichnis

[BF15] D. Bahrdt, S. Funke. „OSCAR: OpenStreetMap Planet at Your Fingertips via OSm Cell ARrangements“. In: *Web Information Systems Engineering – WISE 2015*. Hrsg. von J. Wang, W. Cellary, D. Wang, H. Wang, S.-C. Chen, T. Li, Y. Zhang. Cham: Springer International Publishing, 2015, S. 153–168. ISBN: 978-3-319-26190-4 (zitiert auf S. 4, 7–9, 12).

[Kop17] B. Kopf. „Organisation schwach strukturierter Textdokumente“. 4. Dez. 2017 (zitiert auf S. 4–6, 10–13, 17–19, 32, 33, 35).

[Ley02] M. Ley. „The DBLP Computer Science Bibliography: Evolution, Research Issues, Perspectives“. In: *String Processing and Information Retrieval*. Hrsg. von A. H. F. Laender, A. L. Oliveira. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, S. 1–10. ISBN: 978-3-540-45735-0 (zitiert auf S. 6).

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift