Institut für Formale Methoden der Informatik
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3326

Multimodale Bereichsanfragen im Kontext von Routenplanern

Daniel Bahrdt

Studiengang:	Informatik
Prüfer:	Prof. Dr. Stefan Funke
Betreuer:	Prof. Dr. Stefan Funke
begonnen am:	24. April 2012
beendet am:	25. Oktober 2012
CR-Klassifikation:	H.3.3

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung

Um eine Route zu planen, müssen Start, Ziel und eventuelle Zwischenpunkte bekannt sein. Sind deren Koordinaten nicht bekannt, jedoch andere Informationen, so könnten die Routenpunkte mit Hilfe dieser Informationen gefunden werden. OpenStreetMap bietet hierfür eine interessante Datenbasis, da Geo-Objekte oftmals nicht nur durch Text sondern auch durch strukturierte Informationen beschrieben werden. Ein Supermarkt besitzt dabei neben den Koordinaten noch den Namen, den Typ des Supermarktes sowie eventuell Öffnungszeiten, Internetadressen und mehr. Diese Informationen sollen in dieser Arbeit durch eine Suchmaschinen-ähnliche Texteingabe zugänglich gemacht werden. Die Suche nach textuellen Informationen soll hierbei unter anderem eine Suche nach Teilzeichenketten ermöglichen. Die Ergebnisse der Suche können durch einfache Mengenoperationen miteinander in Verbindung gebracht werden, sodass eine einfache relationale Abfragesprache entsteht. Hauptanwendungsgebiet soll hierbei die Suche auf mobilen Geräten sein. Zu Vergleichszwecken wurde auch ein Programm zur Suche auf einem normalen Desktoprechner entwickelt.

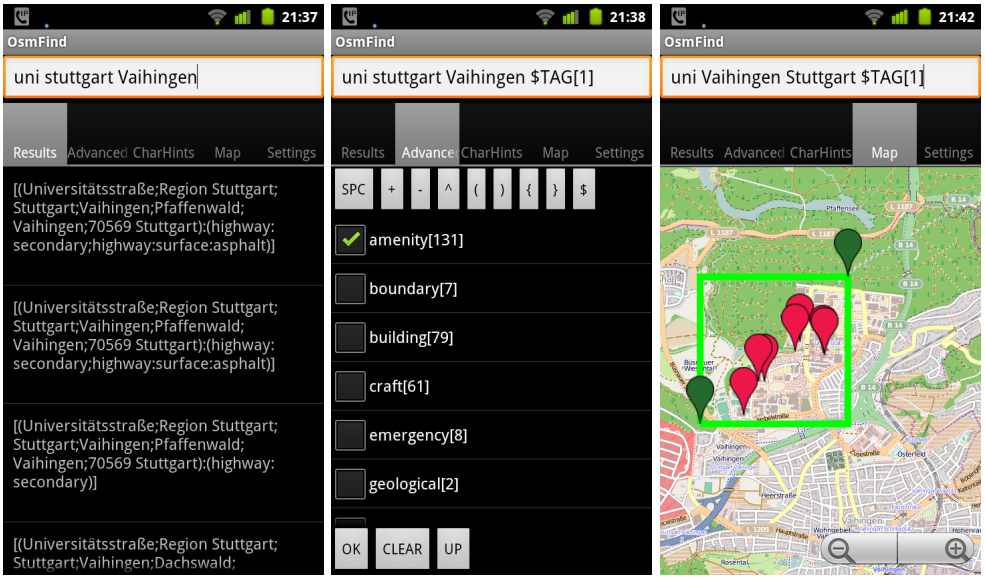


Abbildung 1.1: Links: Komplettierung von Zeichenketten mit Schnitt. Mitte: Auswahlfenster der Tags, Rechts: Erstellen einer Koordinateneinschränkung für die Suche

2 Präliminarien

Sowohl für die Suche in textuellen Daten als auch für die Suche nach Objekten anhand ihrer Koordinaten sind verschiedene Verfahren bekannt und in Verwendung. Zunächst folgt ein kurzer Überblick über mögliche Datenstrukturen und anschließend werden zwei Programme zur Suche in OpenStreetMap-Daten vorgestellt.

2.1 Datenstrukturen und Algorithmen

2.1.1 Textsuche

Suffix-Baum Eine einfache Datenstruktur, Text nach Teilzeichenketten oder Suffixzeichenketten zu durchsuchen, ist ein Suffix-Baum. Hierzu fügt man zu gegebener Quellzeichenkette dessen Suffixzeichenketten in einen Binärbaum ein. Der Speicherplatzbedarf ist hierbei quadratisch, da es zu gegebener Quellzeichenkette genauso viele Suffixzeichenketten gibt, wie die Quellzeichenkette lang ist.

Suffix-Trie Beim Suffix-Trie handelt es sich um eine Baumstruktur, bei der ein Knoten mehrere Kinder besitzen kann, wobei die Kanten zu den Kindknoten beschriftet sind. Zu gegebener Quellzeichenkette erhält man einen Suffix-Trie, indem man alle Suffixzeichenketten in einen Trie einfügt. Ein Pfad von der Wurzel zu einem Blatt gibt hierbei eine Suffixzeichenkette der Quellzeichenkette.

Suffixarray Sowohl im Suffix-Baum als auch im Suffix-Trie müssen neben der Suffixzeichenketten der Quellzeichenkette noch zusätzliche Zeiger für die Elter-Kind-Relationen gespeichert werden. Der Suffixarray, beschrieben in [MM90], umgeht dieses Problem, indem der Baum ohne Zeiger abgespeichert wird. Ein Suffixarray kann man leicht erhalten, indem der Suffix-Baum oder der Suffix-Trie in in-order-Reihenfolge durchlaufen wird. Statt nun jede Zeichenkette zu speichern, kann man auch für jede Suffixzeichenkette den Startpunkt in der Quellzeichenkette speichern. Das so erhaltene Array benötigt somit nur noch linearen Speicherplatz.

Komprimierte Indices Neben den vorgestellten einfachen Index-Strukturen, sind vor allem Kompressions-basierte Indices mit wahlfreiem Zugriff von Interesse. Hier gibt es eine Fülle von unterschiedlichen Algorithmen, die oft auf Suffixarrays aufbauen, mit unterschiedlichen Stärken und Schwächen. Einen Überblick über verschiedene Implementierungen mit Erläuterungen bietet [piz]

2.1.2 Koordinatensuche

Um zu gegebenem Koordinatenintervall alle darin enthaltenen Objekte zu finden, gibt es eine Vielzahl unterschiedlicher Verfahren. Daher sollen nur kurz einige dieser Verfahren angesprochen werden.

Raster Für Datensätze mit begrenzter Ausdehnung bietet sich zunächst ein Raster an, in welches die zu durchsuchenden Objekte abgelegt werden. Diese Datenstruktur eignet sich besonders für Datensätze, die keine großen Leerstellen aufweisen. Dieses Problem lässt sich jedoch zum Teil durch einen Quad-Baum lösen.

Quad-Bäume Im Quad-Baum, beschrieben in [FB74], werden die Daten hierarchisch abgelegt. Eine Rasterzelle wird hierbei nur dann erzeugt, wenn sie belegt ist. Hierdurch können auch Datensätze mit sehr vielen Leerstellen effizient abgespeichert werden.

R-Bäume Gegenüber dem Raster oder dem Quad-Baum ermöglicht der R-Baum das Speichern von Datensätzen mit hoher, theoretisch unbegrenzter, räumlicher Ausdehnung. Hierbei werden zunächst für alle zu speichernden Objekte deren Achsen-parallele minimale Hüllrechtecke bestimmt. Nun können mehrere Objekte zu einem neuen Objekt mit größerem Achsen-parallelen minimalen Hüllrechteck zusammengefasst werden. Die Hüllrechtecke unterschiedlicher Objekte dürfen sich hierbei überlappen. Führt man dieses Verfahren so lange fort, bis nur noch ein zusammengesetztes Objekt existiert, ist ein R-Baum konstruiert. Eine Bereichsanfrage an den R-Baum liefert zunächst mögliche Kandidaten, die genauer untersucht werden müssen. Hierzu beginnt man an der Wurzel des Baumes und prüft für jeden Kindknoten, ob dessen Achsen-paralleles Hüllrechteck mit der Bereichsanfrage kollidiert. Ist dies der Fall, wird die Suche rekursiv fortgesetzt. Enthält der aktuelle Knoten Elemente, die mit der Bereichsanfrage übereinstimmen, werden diese der Ergebnismenge hinzugefügt. Da die Kosten für die Suche von der Anzahl der besuchten Knoten abhängt, ist der Aufbau des Baumes entscheidend für die spätere Suchgeschwindigkeit. Müssen z. B. auf Grund stark überlappender Achsen-paralleler Hüllrechtecke viele Knoten besucht werden, die jedoch gar keine Ergebnisse liefern, verschlechtert dies die Suche massiv. Für den Aufbau des Baumes gibt es daher verschiedene Algorithmen mit verschiedenen Stärken und Schwächen. Ein einfacher Algorithmus ist in [Gut84] beschrieben. Fortgeschrittene Verfahren sind z. B. der R*-Baum von [BKSS90] oder der R+-Baum von [SRF87].

Intervall-Baum Der Intervall-Baum ist eine Baum-Datenstruktur zum Speichern und finden von Intervallen. So lassen sich zu gegebenem Intervall bzw. Punkt alle damit kollidierenden im Baum gespeicherten Intervalle finden.

2.2 Programme

2.2.1 Nominatim

Nominatim von [nom] ist zum Zeitpunkt der Arbeit das Backend für die Suche auf der OpenStreetMap-Seite. Die Suche unterstützt zwei verschiedene Typen. So kann zu gegebenen Koordinaten eine Adresse gefunden werden und zu gegebener Suchzeichenkette passende Objekte. Die Textsuche unterstützt dabei zum Zeitpunkt der Arbeit keine Volltext-Suche. Auch muss die Suchanfrage hierarchisch gestellt werden. So gibt die Suchanfrage *Allmandring Vaihingen Stuttgart* die korrekte Straße zurück, die Suchanfrage *Stuttgart Vaihingen Allmandring* liefert hingegen keine Ergebnisse. Die Sucheingabe wird von einer php-basierten Applikation verarbeitet und beantwortet. Die Daten liegen hierbei in einer relationalen Datenbank vor. Diese Daten werden aus OpenStreetMap-Daten mit Hilfe eines speziellen Programmes in die Datenbank importiert.

2.2.2 Osmand

Osmand, kurz für OSM Automated Navigation Directions, von [osmb] ist eine Navigationssoftware für Android, die OpenStreetMap als Datenbasis nutzt. Osmand kann sowohl mit Online- als auch mit Offline-Daten arbeiten. Der für diese Arbeit interessante Teil ist die Suche nach Objekten. Osmand unterstützt derzeit vier Suchmöglichkeiten. Mit der Point-of-Interest-Suche können nahe gelegene interessante Punkte gefunden werden, wobei eine Vorselektierung der möglichen PoI durchgeführt wurde. Darüberhinaus bietet es die Möglichkeit, einen PoI über den Namen zu finden. Die PoI-Suche beschränkt sich hierbei auf eine vom Nutzer einstellbare Umgebung um einen Punkt. Im Gegensatz zur PoI-Suche ist die Adresssuche keine Umkreissuche. Ähnlich einem Autonavigationsgerät kann eine Adresse durch eine hierarchische Suche gefunden werden. Zunächst muss die Region, in der gesucht werden soll, ausgewählt werden. Dann folgt die Stadt und zuletzt die Straße. Gesucht werden dabei diejenigen Elemente, deren Präfix mit der Eingabe übereinstimmen. Eine Teilzeichenkettensuche oder komplexere Suchoperationen sind nicht möglich. Allerdings kann, falls eine Internetverbindung besteht, Nominatim als Suche genutzt werden.

3 OpenStreetMap

3.1 Einleitung

[osma]: "OpenStreetMap is a free worldwide map, created by people like you." OpenStreetMap ist eine freie Geodatenbank, die von Nutzern auf der ganzen Welt mit Hilfe von GPS-Geräten und Orthofotos erstellt wird. Die Daten stehen unter der [ODb], einer freien Datenbanklizenz. OpenStreetMap wurde 2004 von Steve Coast ins Leben gerufen. Seither haben sich die Nutzerzahl und die Datenbestände massiv vergrößert. In einigen Regionen übersteigt der Detailgrad der Geodaten den kommerzieller Anbieter, was sich leicht mit Hilfe von [geob] oder [osmc] verifizieren lässt.

3.2 Datentypen

Derzeit gibt es drei Grunddatentypen, die jedoch durch zusätzliche Attribute, die als Schlüssel-Wert-Paare abgelegt werden, augmentiert werden können. Der einfachste Datentyp ist der Knoten **node**. Dabei handelt es sich im einfachsten Fall um einen Punkt mit Koordinaten im WGS84-Bezugssystem und einer eindeutigen Nummer (node-id). In Verbindung mit entsprechenden Attributen kann z.B. ein Banknotenautomat als Point-of-Interest erfasst werden. Um Knoten miteinander verknüpfen zu können, gibt es den gesonderten Datentyp **way**. Dieser vereint Knoten zu einem ungerichteten Kantenzug. Für komplexere Zusammenhänge, die auch ineinander verschachtelt werden können, wird der **relation** Datentyp genutzt, womit z.B. mehrere Wege zu einer Wanderroute zusammen gefasst werden können. Attribute beschreiben hierbei den genauen Typ der Relation zwischen den enthaltenen Objekten. Tabelle 3.1 enthält einige Beispiele zur Kodierung, Tabelle 3.2 zeigt einige Beispiele zur Definition von komplexeren Schlüssel-Wert-Hierarchien.

3.3 Dateiformate

Die im Abschnitt 3.2 vorgestellten Datentypen werden in verschiedenen Dateiformaten bereit gestellt. Intern verwendet OpenStreetMap eine relationale Datenbank. Von dieser können Kartenausschnitte über eine Web-API angefordert werden. Hierbei werden die Datentypen in einem xml-Dokument kodiert zurückgegeben. Darüberhinaus wird in regelmäßigen Abständen die gesamte Datenbank als XML-Datei abgespeichert und zur Verfügung gestellt.

	Datentyp	Attribute
Arztpraxis	node	amenity=doctors
Apotheke	node	amenity=pharmacy
Autobahn	way	highway=motorway
Straße mit Fahrradweg	way	highway=primary, cycleway=lane
Gebäude	way	area=yes, building=yes
Wanderroute	relation	type=route, route=hiking

Tabelle 3.1: Ausgewählte Beispiele zur Kodierung in OpenStreetMap

Schlüssel	Bedeutung
name	Name in der Landessprache
name:de	Name auf Deutsch
name:fr	Name auf Französisch
cycleway:surface	Explizite Angabe der Bodenbeschaffenheit eines Fahrradweges, der zusätzlich zu einer vorhandenen Straße mit anderer Bodenbeschaffenheit verläuft

Tabelle 3.2: Beispiel-Schlüssel-Werte in OpenStreetMap

Abbildung 3.1 zeigt einen Ausschnitt eines solchen xml-Dokuments. Neben dem XML-Format gibt es noch weitere Dateiformate, von welchen das OpenStreetMap-ProtocolBuffers-Dateiformat das am weitesten verbreitete ist. Im Gegensatz zum XML-Format, welches die Daten als Text kodiert, handelt es sich hierbei um ein Binärformat, wodurch die Dateigröße massiv abnimmt. Eine genaue Beschreibung des Dateiformates sowie die in dieser Arbeit genutzte Abstraktionsbibliothek findet sich in [Gro12]. Die für diese Arbeit verwendeten Daten entstammen alle den Kartenausschnitten der Geofabrik, welche in regelmäßigen Abständen den Weltdatensatz in kleinere Datensätze getrennt nach Regionen, Ländern und teilweise Bundesländern, unter [geoa] zum Herunterladen, bereitstellt.

	XML-Format	OSM-PBF-Format
Europa	167.388 MiB	8.937 MiB
Frankreich	49.648 MiB	2.149 MiB
Deutschland	22.738 MiB	1.322 MiB
Großbritannien	8500 MiB	2.149 MiB
Spanien	6240 MiB	307 MiB
Italien	11.290	498 MiB
Niederlande	10.341 MiB	506 MiB
Baden-Württemberg	3370 MiB	189 MiB

Tabelle 3.3: Dateigrößen einiger Länderausschnitte der Geofabrik

Listing 3.1 Beispiel des XML-Formats mit Knoten, Wegen und Relationen

```
<osm version="0.6">
    <bound box="-90,-180,90,180" />
    <node id="270387" lat="48.482063" lon="9.367616" >
        <tag k="name" v="Wasserfall Bad Urach" />
    </node>
    <way id="99947113">
        <nd ref="1155236845" />
        <nd ref="1155223179" />
        <nd ref="1155236845" />
        <tag k="building" v="yes" />
        <tag k="wall" v="no" />
    </way>
    <relation id="1430044">
        <member type="way" ref="27549584" role="street"/>
        <member type="node" ref="1155672139" role="house"/>
        <member type="node" ref="1155672022" role="house"/>
        <member type="node" ref="1155672148" role="house"/>
        <tag k="name" v="Rue Geoffroy-Drouet" />
        <tag k="type" v="associatedStreet" />
    </relation>
</osm>
```

3.4 Nutzung

Für die Textsuche werden das name-Attribut sowie die Grenzpolygone verwendet. Letztere werden mit den Geo-Objekten geschnitten, um so deren Informationen den Geo-Objekten zuzuweisen. Darüberhinaus werden auch die Attribute verwendet, die jedoch auch eine zusätzliche Behandlung erfordern. Da es sich bei OpenStreetMap um ein Gemeinschaftsprojekt handelt, werden jedoch ständig neue Attribute erfunden, um z. B. ein noch nicht definiertes Objekt zu beschreiben. Daher gibt es sehr viele Attribute, die (bisher) keine breite Verwendung gefunden haben. Für diese Arbeit wurde daher eine Vorselektierung der Attribute gemacht. Tabelle 3.4 gibt dabei einen Überblick über interessante Attribute, die einen sehr hohen Verbreitungsgrad besitzen.

Schlüssel	Unterkategorie	Beschreibung
amenity	Wurzel	Wichtige bzw. für Anwohner und Touristen interessante Orte
boundary	Wurzel	Grenze (z.B. Stadtteilgrenze)
building	Wurzel	Gebäudetyp
craft	Wurzel	Geschäft für handwerkliche Güter
emergency	Wurzel	Unterschiedliche Notfalleinrichtungen
geological	Wurzel	geologische Sehenswürdigkeit/Begebenheit
highway	Wurzel	Straßentyp (z.B. Autobahn oder Spielstraße)
historic	Wurzel	Objekt von historischem Interesse (z.B. eine Burg)
landuse	Wurzel	Nutzung eines Landstücks
leisure	Wurzel	Freizeiteinrichtungen
man_made	Wurzel	menschliche Bauwerke (z.B. eine Windmühle)
military	Wurzel	militärische Einrichtungen
natural	Wurzel	Natürliche Begebenheiten (z.B. ein Strand)
office	Wurzel	Geschäft, das Dienstleistungen anbietet
place	Wurzel	Definiert den Typ einer Siedlung (Land / Stadt / Dorf)
public_transport	Wurzel	Öffentliche Verkehrsmittel
railway	Wurzel	Schienen
route	Wurzel	Eine Route (z.B. Fahrradwege, Busrouten)
shop	Wurzel	Ladengeschäft
sport	Wurzel	Sportstätte
tourism	Wurzel	touristisch interessantes Objekt
waterway	Wurzel	Wasserwege (Fluss, Bach)
cycleway	highway	Radweg
sac_scale	highway	Schwierigkeitsgrad eines Weges
surface	highway	Bodenbeschaffenheit
tracktype	highway	Genauere Wegklassifizierung
cuisine	amenity:restaurant	kulinarische Einordnung eines Restaurants

Tabelle 3.4: Schlüssel, die für die Tagsuche interessant sind sowie ihre Einordnung in eine Hierarchie

4 Datenstrukturen und Algorithmen

4.1 Einführung

In diesem Kapitel sollen die wichtigsten verwendeten Datenstrukturen und Algorithmen vorgestellt werden. Zunächst werden die Datenstrukturen, die zur Verarbeitung eingesetzt wurden, vorgestellt. Anschließend werden anhand des Weges einer Suchanfrage die dabei genutzten Datenstrukturen näher erläutert.

4.2 Vorverarbeitung

4.2.1 Polygonschneiden

Um zu den OpenStreetMap-Objekten weitere Informationen hinzuzufügen, werden diese mit Grenzpolygonen geschnitten. Da diese Schnittoperationen sehr oft durchgeführt werden müssen, sollte die Datenstruktur Punkt-in-Polygon-, Linie-Polygon-Kollision- und Polygon-Polygon-Kollision-Anfragen effizient beantworten können. Einen Überblick über mögliche Datenstrukturen gibt Abschnitt 2.1.2. Für diese Arbeit wurde ein Koordinatenraster genutzt, wobei in jeder Zelle all jene Polygone gespeichert werden, die mit dieser kollidieren oder die Zelle vollständig umschließen. Um eine Anfrage zu beantworten, müssen zunächst alle relevanten Gitterzellen ermittelt werden. Dies geschieht mit Hilfe des Grenzrechtecks der Eingabe. Je nach Eingabe-Typ müssen die so erhaltenen Gitterzellen unterschiedlich weiterverarbeitet werden. Um Punkt-in-Polygon-Anfragen zu beantworten, müssen lediglich die Polygone, die mit der Gitterzelle kollidieren, getestet werden. Polygone, die die Gitterzelle umschließen, können ohne weitere Prüfung übernommen werden. Für Wege und Polygone werden zunächst die Grenzrechtecke getestet, anschließend die einzelnen Wegpunkte. Liefert dies kein positives Ergebnis, so müssen noch die Kollisionen mit den einzelnen Segmenten getestet werden. Die Daten der so erhaltenen Grenzpolygone werden dem OpenStreetMap-Objekt hinzugefügt.

4.2.2 Generalisierter Suffix-Trie

Der Generalisierte Suffix-Trie ist eine Baumdatenstruktur zur effizienten Suche nach Suffix-Zeichenketten. Hierzu werden alle Suffixe einer Zeichenkette in einen Trie eingefügt, wobei Knoten mit nur einem Kind zusammengefasst werden. Einen Generalisierten Suffix-Trie, in welchem man verschiedene Zeichenketten speichern kann, kann man erhalten, indem man

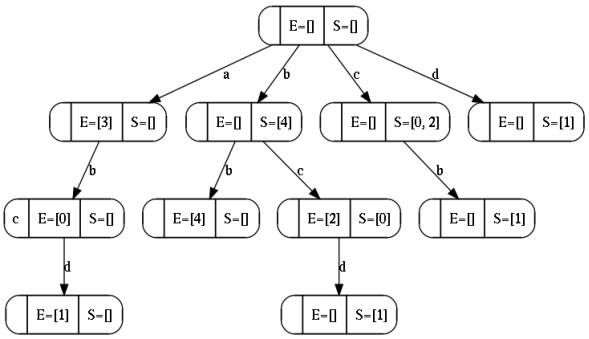


Abbildung 4.1: Generalisierter Suffix-Trie für die Zeichenketten abc=0, abcd=1, bc=2, a=3, bb=4. Endmengen werden mit E für die exakten Zeichenkettenmengen und S für Suffixzeichenkettenmengen abgekürzt

für jede Zeichenkette ein eigenes Zeichenkettenende-Symbol verwendet. Anhand dieses Zeichenkettensymbols können die Suffixe der einzelnen Zeichenketten im Baum unterschieden werden. Konkret wurde diese Datenstruktur in dieser Arbeit wie folgt umgesetzt: Kanten erhalten nur ein Zeichen, Knoten hingegen können auch mehrere Zeichen enthalten, sofern sie nur einen Kindknoten hätten. Knoten können sowohl innere Knoten als auch Endknoten sein. Hierzu besitzt jeder Knoten eine Liste von Zeigern auf diejenigen Zeichenketten, welche diesen Knoten als ein Ende eines ihrer Suffixe haben. Um unterscheiden zu können, ob es sich um das Ende der ganzen Zeichenkette oder nur eines Suffixes handelt, gibt es eine Liste für die Suffixenden und eine für die exakten Enden. Hierdurch ist es möglich, im Baum nach einer Exakten-, Präfix-, Suffix- oder Teilzeichenketten-Übereinstimmung zu suchen. Eine Suche, die die Groß-/Kleinschreibung nicht beachtet kann entweder dadurch erreicht werden, dass die Zeichenketten ohne Großbuchstaben gespeichert werden, oder indem beim Abstieg der Pfad für die Groß-, als auch für die Kleinschreibung verfolgt wird. Selbiges gilt analog für diakritische Zeichen.

4.3 Anfrageverarbeitung

Eine Suchanfrage beginnt zunächst bei der Eingabe des Nutzers. Dies spezifiziert die genaue Komplettierung. Für diese Arbeit wurde hierfür eine Abfragesprache ähnlich der von Suchmaschinen gewählt. Die Suchanfrage wird nun mit Hilfe dieser Grammatik analysiert und in eine nutzbare Datenstruktur überführt. Dies ist der Mengenoperationsbaum auf welchem anschließend sämtliche Komplettierungs- und Mengenoperationen durchgeführt werden. Anschließend wird die Ergebnismenge dem Nutzer präsentiert.

4.3.1 Mengenoperationsbaum

Der Mengenoperationsbaum ist die Datenstruktur, um die Mengenoperationen der Suchanfrage durchzuführen. Er ermöglicht die Aktualisierung der Suchanfrage, wobei hierbei

lediglich die Teile neu berechnet werden, die sich geändert haben. Hierdurch werden Suchanfragen, die sukzessive die Ergebnismengen reduzieren, erheblich schneller. Je nach Anwendungsszenario können die Mengenoperationen vollständig durchgeführt werden, oder aber eine Iterator-Schnittstelle genutzt werden. Hierbei werden die Operationen sukzessive ausgeführt.

4.3.2 Mengendatenstruktur

Zentral für die gesamte Anfrageverarbeitung ist eine Datenstruktur, die die üblichen Mengenoperationen effizient unterstützt. In dieser Arbeit wurde daher ein sortierter Array zur Darstellung einer Komplettierungsmenge gewählt. Hierdurch können die üblichen Mengenoperationen wie Vereinigung, Schnitt, Differenz sowie symmetrische Differenz in linearer Zeit durchgeführt werden. Für Mengen sehr unterschiedlicher Größe kann eine Verbesserung durch die Nutzung einer Binärsuche erzielt werden. Statt $O(|M1| \cdot |M_2|)$ werden nur $O(|M1| \cdot \log_2(|M2|))$ Vergleiche benötigt. Der Einsatz bleibt dabei jedoch auf den Schnitt und die Mengendifferenz beschränkt. Für den Iterator-basierten Zugriff erfolgen sämtliche Mengenoperationen in linearer Zeit mit Hilfe eines Merge-Algorithmus. Falls viele Mengen miteinander vereinigt werden müssen, so wird eine Baum-basierte Variante genutzt, welche in Abbildung 4.5 genauer beschrieben wird. Darüberhinaus kann auch eine Bit-Vektor-basierte Speicherung erfolgen. Hierbei bedeutet ein gesetztes Bit an Stelle i, dass Element i Teil der Menge ist. Mengen-Operationen können sodann in linearer Zeit durch einfache Bit-Operationen durchgeführt werden. Um Speicherplatz zu sparen, kann z.B. eine Laufweitenkodierung verwendet werden. Eine weitere Möglichkeit ist die Speicherung der Mengen durch Differenzkodierung zum jeweils vorhergehenden Element. Sowohl bei der Bit-Vektor-basierten Speicherung mit Laufweitenkodierung als auch bei der Differenzkodierung ist ein wahlfreier Zugriff unmöglich, wodurch z. B. eine Binärsuche nicht genutzt werden kann.

4.4 Textsuche

4.4.1 Statischer GST mit Zeigern

Da auf mobilen Geräten eine Erzeugung der Datenstrukturen nicht nötig bzw. zu langsam ist, kann eine Datenstruktur gewählt werden, die lediglich Leseoperationen ermöglicht. Darüberhinaus sollte sich der Arbeitsspeicherbedarf in Grenzen halten, da z.B. unter Android eine Anwendung nicht mehr als 20 Megabyte Arbeitsspeicher benötigen sollte. Hierzu wurde der Generalisierte Suffix-Trie serialisiert. Die Zeiger auf die Kindknoten sind entsprechende Offsets in der Baumdatei. Diese sind entsprechend ihrer Kantenbeschriftung aufsteigend sortiert abgespeichert, wodurch zu gegebenem Zeichen in $O(\log_2(n))$ der zugehörige Zeiger gefunden werden kann. Zu gegebener Komplettierungszeichenkette der Länge m kann so in $O(m \cdot \log_2(|Alphabet|))$ der Endknoten gefunden werden. Für die Präfix- und Teilzeichenkettensuche muss zu gegebenem Endknoten dessen Unterbaum komplett durchlaufen

Listing 4.1 Merge-Algorithmus für die Vereinigung

```
1  Ergebnismenge Vereinigung(MengeA, MengeB) {
2    i=0; j=0;
3    while(i < Größe(MengeA) && j < Größe(MengeB)) {
4      if (MengeA[i] == MengeB[j]) {
5      Füge MengeA[i] an das Ende von Ergebnismenge hinzu;
6        ++i;
7        ++j;
8      }
9      else if (MengeA[i] < MengeB[j]) {
10       Füge MengeA[i] an das Ende von Ergebnismenge hinzu;
11       ++i
12     }
13     else {
14       Füge MengeB[j] an das Ende von Ergebnismenge hinzu;
15       ++j;
16     }
17   }
18   Füge Elemente zwischen i und Größe(MengeA) zur Ergebnismenge hinzu
19   Füge Elemente zwischen j und Größe(MengeB) zur Ergebnismenge hinzu
20 }
```

Listing 4.2 Merge-Algorithmus für den Schnitt

```
1  Ergebnismenge Schnitt(MengeA, MengeB) {
2    i=0; j=0;
3    while(i < Größe(MengeA) && j < Größe(MengeB)) {
4      if (MengeA[i] == MengeB[j]) {
5      Füge MengeA[i] an das Ende von Ergebnismenge hinzu;
6        ++i;
7        ++j;
8      }
9      else if (MengeA[i] < MengeB[j]) {
10       ++i
11     }
12     else {
13       ++j;
14     }
15   }
16 }
```

werden. Um dies zu beschleunigen, kann in der serialisierten Variante die Menge der Zeichenkettenzeiger des Unterbaumes auch im Knoten gespeichert werden. Hierzu wurden mehrere Speichervarianten verglichen.

Direkte Komplettierungsmengen

Die einfachste Möglichkeit ist die direkte Abspeicherung der Mengen. Dabei werden die Exakte-, Präfix-, Suffix- und Teilzeichenkettenmenge für jeden Knoten vollständig abgespeichert.

Listing 4.3 Merge-Algorithmus für die Mengendifferenz

```
1  Ergebnismenge Differenz(MengeA, MengeB) {
2    i=0; j=0;
3    while(i < Größe(MengeA) && j < Größe(MengeB)) {
4      if (MengeA[i] == MengeB[j]) {
5        ++i;
6        ++j;
7      }
8      else if (MengeA[i] < MengeB[j]) {
9        Füge MengeA[i] an das Ende von Ergebnismenge hinzu;
10       ++i
11     }
12     else {
13       ++j;
14     }
15   }
16   Füge Elemente zwischen i und Größe(MengeA) zur Ergebnismenge hinzu
17 }
```

Listing 4.4 Merge-Algorithmus für die symmetrische Mengendifferenz

```
1  Ergebnismenge SymmetrischeDifferenz(MengeA, MengeB) {
2    i=0; j=0;
3    while(i < Größe(MengeA) && j < Größe(MengeB)) {
4      if (MengeA[i] == MengeB[j]) {
5        ++i;
6        ++j;
7      }
8      else if (MengeA[i] < MengeB[j]) {
9        Füge MengeA[i] an das Ende von Ergebnismenge hinzu;
10       ++i
11     }
12     else {
13       Füge MengeB[j] an das Ende von Ergebnismenge hinzu;
14       ++j;
15     }
16   }
17   Füge Elemente zwischen i und Größe(MengeA) zur Ergebnismenge hinzu
18   Füge Elemente zwischen j und Größe(MengeB) zur Ergebnismenge hinzu
19 }
```

Listing 4.5 Baum-basierter Merge-Algorithmus für die Vereinigung

```
1  Ergebnismenge BaumVereinigung(Array A aus Mengen, Start, Ende) {
2    if (Start == Ende) {
3      Ergebnismenge = A[Start]
4    }
5    else {
6      Mitte = (Ende-Start)/2 + Start
7      MengeLinks = BaumVereinigung(A, Start, Mitte)
8      MengeRechts = BaumVereinigung(A, Mitte+1, Ende)
9      Ergebnismenge Vereinigung(MengeLinks, MengeRechts)
10   }
11 }
```

Indirekte Komplettierungsmengen

Zwischen der Exakten-/Suffixzeichenkettenmenge der Knoten des Unterbaumes und der Präfix- bzw Teilzeichenkettenmenge besteht eine Teilmengenbeziehung. Diese kann genutzt werden, um ein Remapping der Zeiger in den Exakten-/Suffixzeichenkettenmenge durchzuführen. Die Zeiger in der Exakten-/Suffixzeichenkettenmenge bezieht sich dann auf die Position in der Präfix- bzw. Teilzeichenkettenmenge des Elterknotens. Hierdurch werden die Zeiger kleiner und deren Verteilung homogener, was eine bessere Approximierung durch eine Regressionsgerade ermöglicht.

Dynamische Komplettierungsmengen

Statt alle Präfix- und Teilzeichenkettenmengen zu speichern, kann die Speicherung derselben anhand verschiedener Kriterien wie etwa der Ebene im Baum oder die Anzahl der nötigen Vergleiche zur Erzeugung der Mengen, bestimmt werden. Darüberhinaus kann man sich noch zu Nutze machen, dass gilt:

$$\begin{aligned}
\textbf{E}\text{xaktezeichenkettenmenge}_{Knoten} &\subseteq \textbf{P}\text{räfixzeichenkettenmenge}_{Knoten} \\
\textbf{E}\text{xaktezeichenkettenmenge}_{Knoten} &\subseteq \textbf{S}\text{ufffixzeichenkettenmenge}_{Knoten} \\
\textbf{P}\text{räfixzeichenkettenmenge}_{Knoten} &\subseteq \textbf{T}\text{eilzeichenkettenmenge}_{Knoten} \\
\textbf{S}\text{uffixzeichenkettenmenge}_{Knoten} &\subseteq \textbf{T}\text{eilzeichenkettenmenge}_{Knoten}
\end{aligned}$$

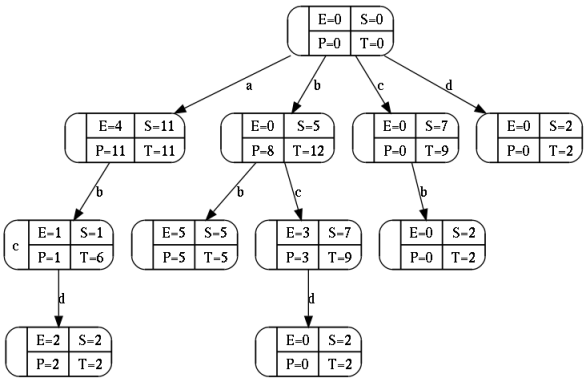
Gespeichert wird somit nur $E = E$, $P = P - E$, $S = S - E$ und $T = T - E - P - S$. Einher geht hiermit jedoch ein entsprechender Geschwindigkeitsverlust, da die Mengen für die korrekte Komplettierung wieder zusammengeführt werden müssen.

4.4.2 Flacher Generalisierter Suffix-Trie

Der flache GST, kurz FlatGST, kann ebenfalls aus dem GST erzeugt werden. Dieser ist im Grunde ein Suffixarray mit explizit gespeicherten inneren Knoten. Dieser entsteht, indem der GST in in-order-Reihenfolge durchlaufen wird. Dabei werden für jeden Knoten dessen Pfad von der Wurzel bis zum Knoten gespeichert. Da der Baum im Falle eines Suffix-Trie aus Suffixen von Zeichenketten entstanden ist, ist es nicht nötig, für jeden Knoten die vollständige Suffixzeichenkette abzuspeichern. Stattdessen speichert man für jeden Knoten einen Zeichenkettenzeiger, die Startposition in dieser Zeichenkette sowie die Länge des Knotenpfades. Wie im normalen GST müssen auch im flachen GST die Knotenmengen für die Exakte, Präfix, Suffix sowie Teilzeichenketten-Übereinstimmungen gespeichert werden.

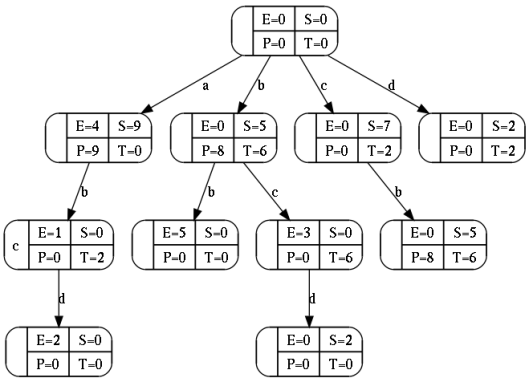
Suche

Um zu gegebener Suchzeichenkette einen eventuell vorhandenen Endknoten zu finden, kann eine einfache Binärsuche genutzt werden. Da hierbei der Bereich immer stärker eingeengt



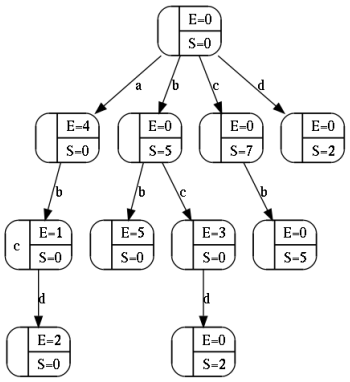
Mengenid	Menge
0	{}
1	{0}
2	{1}
3	{2}
4	{3}
5	{4}
6	{0,1}
7	{0,2}
8	{2,4}
9	{0,1,2}
11	{0,1,2,3}
12	{0,1,2,4}

Abbildung 4.2: Generalisierter Suffix-Trie für die Zeichenketten abc=0, abcd=1, bc=2, a=3, bb=4 mit vollem Index. Endmengenzeiger werden mit E für die Exakte-, P für Präfix-, S für Suffix- und T für Teilzeichenkettenrelationen bezeichnet



Mengenid	Menge
0	{}
1	{0}
2	{1}
3	{2}
4	{3}
5	{4}
6	{0,1}
7	{0,2}
8	{2,4}
9	{0,1,2}

Abbildung 4.3: Generalisierter Suffix-Trie für die Zeichenketten abc=0, abcd=1, bc=2, a=3, bb=4 mit Merge-Index. Endmengenzeiger werden mit E für die Exakte-, P für Präfix-, S für Suffix- und T für Teilzeichenketten-Übereinstimmung bezeichnet



Mengenid	Menge
0	$\{\}$
1	$\{0\}$
2	$\{1\}$
3	$\{2\}$
4	$\{3\}$
5	$\{4\}$
7	$\{0,2\}$

Abbildung 4.4: Generalisierter Suffix-Trie für die Zeichenketten abc=0, abcd=1, bc=2, a=3, bb=4 mit Merge-Index ohne Präfix- und Teilzeichenkettenindex. Endmengenzeiger werden mit E für die Exakte und P für Präfixzeichenketten-Mengen bezeichnet

wird, haben die Zeichenketten des Suchbereichs mit der Suchtiefe ein immer längeres gemeinsames Präfix. Dies kann dazu genutzt werden, die Anzahl der Vergleiche zu verringern. Hierzu muss am Anfang der Suche das gemeinsame Präfix zwischen der Suchzeichenkette und den Intervallgrenzen bekannt sein. Sodann bekommt man durch den Vergleich mit dem Intervallmittenelement ein weiteres gemeinsames Präfix. Steigt man nun ins linke Intervall ab, so müssen nur diejenigen Zeichen der Suchzeichenkette untersucht werden, die nicht Teil des gemeinsamen Präfix aus linker Intervallgrenze und Intervallmittenelement sind. Analoges gilt für den Abstieg in die rechte Hälfte. Listing 4.6 zeigt den Algorithmus in Pseudocode.

Zeichenkettenreferenzen

In Verbindung mit einer Datenbank, die die zu durchsuchenden Objekte enthält, kann die Speichermenge reduziert werden, indem nicht die Objektreferenz im Baum erfasst wird, sondern lediglich dessen Zeichenkettenreferenzen. Sodann kann die Datenbank nach den Zeichenkettenreferenzen durchsucht werden, was $O\left(n \cdot m \cdot \log k\right)$ Zeit benötigt (n = Datenbankeinträge, m = Anzahl der Zeichenkettenreferenzen je Eintrag, k = Anzahl der Zeichenkettenreferenzen der Komplettierungsmenge). Der Zugriff kann auf Kosten höheren Speicherplatzes bis zum normalen flachen GST beschleunigt werden, indem die Endknotenmengen-Typen von der Größe derselben abhängen. Dabei werden nach einem vorher festgelegten Kriterium Zeichenkettenereferenz-Mengen oder Objektreferenz-Mengen gespeichert. Wählt man als Kriterium die Größe der Objektreferenz-Menge und die Länge des Knotenpfades und nutzt die Iterator-Schnittstelle, so lässt sich die Speichermenge ohne großen Geschwindigkeitsverlust

Listing 4.6 Algorithmus zur Suche im flachen Generalisierten Suffix-Trie

1. Eingabe von LinkesGemeinsamesPräfix und RechtesGemeinsamesPräfix, LinkeGrenze, RechteGrenze
2. Berechnung von MittleresGemeinsamesPräfix ab Minimum(LinkesGemeinsamesPräfix, RechtesGemeinsamesPräfix)
3. Entscheid für linken Abstieg
   - RechtesGemeinsamesPräfix = MittleresGemeinsamesPräfix
   - RechteGrenze = Mitte
4. Entscheid für rechten Abstieg
   - RechtesGemeinsamesPräfix = MittleresGemeinsamesPräfix
   - LinkeGrenze = Mitte
5. Entscheid für Ende
6. Gehe zu 0

Zeichen	Id	Beginn	Länge	Endmengen
a	0	0	1	$E = \{3\}$, $P = S = T = \{0,1,2,3\}$
abc	0	0	3	$E = P = S\,\{0\}$, $T = \{0,1\}$
abcd	1	0	4	$E = P = S = T = \{1\}$
b	0	1	1	$E = \{\}$, $P = 2,4$, $S4$, $T = \{0,1,2,4\}$
bb	4	0	2	$E = P = S = T = \{4\}$
bc	0	1	2	$E = \{2\}$, $P = \{2\}$, $S = \{0,2\}$, $T = \{0,1,2\}$
bcd	1	1	3	$E = P = \{\}$, $S = T = \{1\}$
c	0	2	1	$E = P\,\{\}$, $S = \{0,2\}$, $T = \{0,1,2\}$
cd	1	2	2	$E = P = \{\}$, $S = T = \{1\}$
d	1	3	1	$E = P = \{\}$, $S = T = \{1\}$

Abbildung 4.5: Flacher Generalisierter Suffix-Trie für die Zeichenketten abc=0, abcd=1, bc=2, a=3, bb=4. Endmengen werden mit E für die Exakte-, P für Präfix-, S für Suffix- und T für Teilzeichenketten-Übereinstimmung bezeichnet

Zeichen	Id	Beginn	Länge	Endmengen
a	0	0	1	$E = \{3\}, P = \{0,1,2\}, S = \{0,1,2\}, T = \{\}$
abc	0	0	3	$E = \{0\}, P = S\,\{\}, T = \{1\}$
abcd	1	0	4	$E = \{1\}, P = S = T = \{\}$
b	0	1	1	$E = \{\}, P = \{2,4\}, S = \{4\}, T = \{0,1\}$
bb	4	0	2	$E = \{4\}, P = S = T = \{\}$
bc	0	1	2	$E = \{2\}, P = \{\}, S = \{\}, T = \{0,1\}$
bcd	1	1	3	$E = P = \{\}, S = \{1\}, T = \{\}$
c	0	2	1	$E = P\,\{\}, S = \{0,2\}, T = \{1\}$
cd	1	2	2	$E = P = \{\}, S = \{1\}, T = \{\}$
d	1	3	1	$E = P = \{\}, S = \{1\}, T = \{\}$

Abbildung 4.6: Flacher Generalisierter Suffix-Trie wie in Abbildung 4.5 aber mit einem Merge-Index

reduzieren, indem sehr große Objektreferenz-Mengen durch Zeichenkettenreferenz-Mengen ersetzt und erst ab einer bestimmten Pfadlänge Objektreferenz-Mengen gespeichert werden.

4.4.3 Lineare Suche

Die einfachste Variante, alle Dokumente zu gegebener Suchzeichenkette zu finden, ist das Durchsuchen der Datenbank. Die Suchkosten betragen hierfür im schlechtesten Fall $O\left(s \cdot m + k \cdot \log k\right)$ (s = Anzahl unterschiedlicher Zeichenketten, m = Länge der Zeichenketten, k = Anzahl der übereinstimmenden Zeichenketten). Anschließend müssen noch wie im FlatGST mit Zeichenkettenreferenzen die richtigen Objekte gefunden werden.

4.5 Tag-Suche

Für die Tag-Suche wurde ebenfalls ein Baum-basierter Ansatz gewählt. Die Tags können hierarchisch gruppiert werden, was im Grunde einen Radix-Trie ergibt, wobei die Alphabetmenge die Schlüssel-Wert-Paare der Tags sind. Abbildung 4.7 zeigt einen Tag-Trie mit einer Tag-Hierarchie. Die Eingabe erfolgt über ein spezielles Schlüsselwort, wobei die gewünschten Tags entweder über ihren Pfad im Baum oder über die direkte Angabe des Knotens selektiert werden können.

4.6 Koordinatensuche

Die Koordinatensuche ermöglicht die Suche nach Elementen, die sich innerhalb eines Koordinatenintervalls befinden. Falls bereits eine Komplettierung vorliegt, so können die

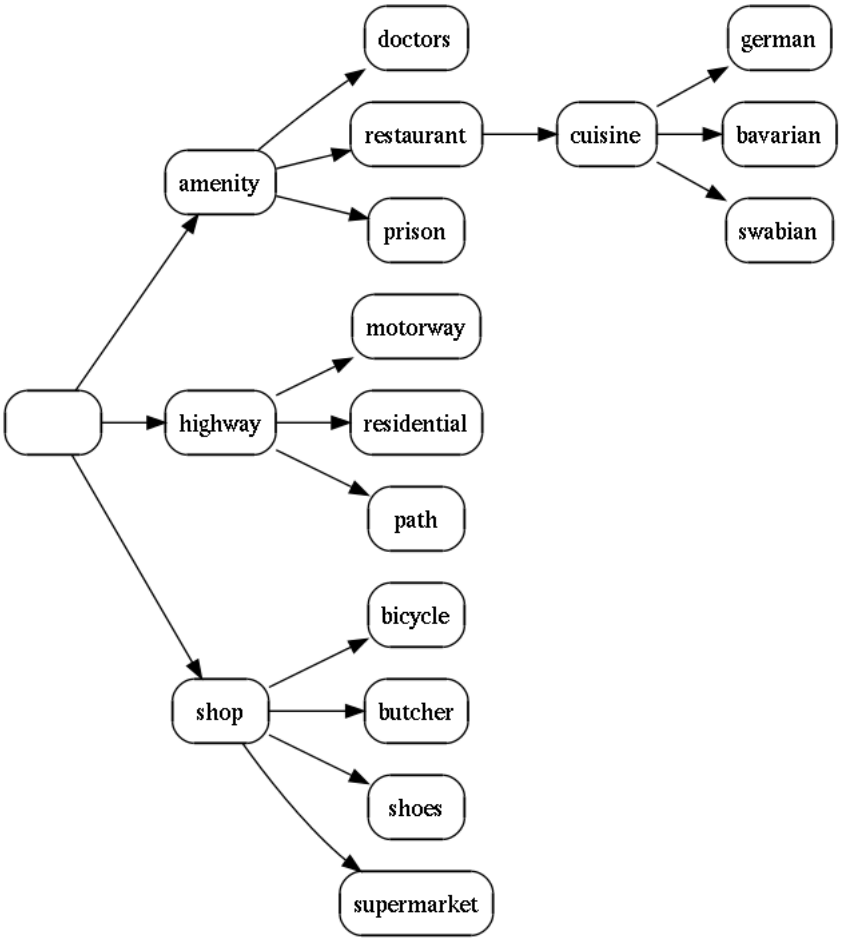


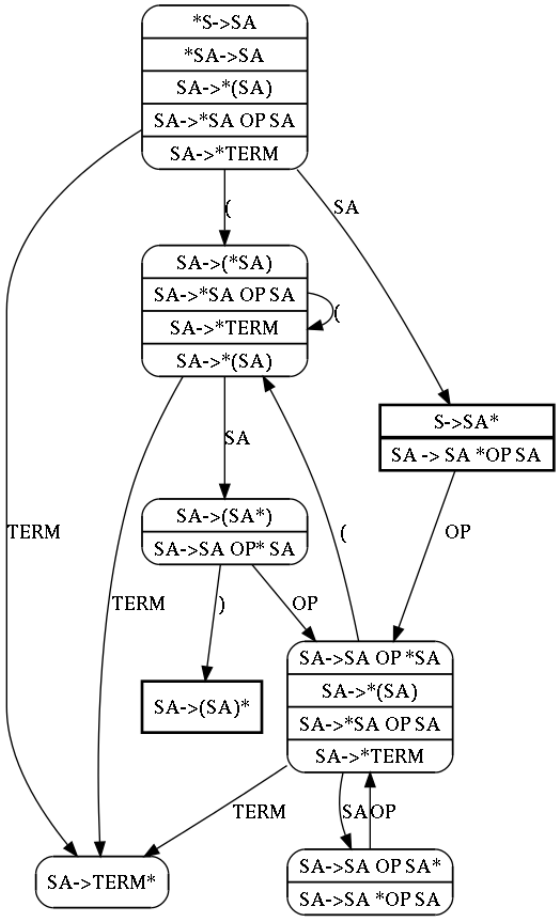
Abbildung 4.7: Beispiel-Trie mit verschiedenen Tags

Elemente in linearer Zeit nach den Koordinatenintervallen gefiltert werden. Liegt noch keine Komplettierung vor, muss auf andere Geodatenstrukturen zurückgegriffen werden. Neben einem einfachen Koordinatenraster bieten sich auch Baum-basierte Algorithmen an. In dieser Arbeit wurde auf Grund der Einfachheit zunächst ein Koordinatenraster erstellt. Darüberhinaus können im Gitter die Elemente in den einzelnen Gitterzellen in sortierter Reihenfolge abgelegt werden, was die weitere Verarbeitung erheblich vereinfacht.

Um zu gegebenen Koordinatenintervallen alle Elemente zu erhalten, müssen zunächst alle Gitterzellen, die mit den Koordinatenintervallen kollidieren ermittelt werden. Hierbei gibt es Gitterzellen, die vollständig im Koordinatenintervall liegen, und Gitterzellen, die die Koordinatenintervalle nur schneiden. Bei letzteren Gitterzellen muss daher ein weiterer Filter zum Aussortieren falscher Elemente verwendet werden. Anschließend können alle Gitterzellen mit einem Baum-basierten Merge-Algorithmus vereinigt werden.

4.7 Sprachdefinition

Die Eingabe der drei möglichen Suchkriterien erfolgt über normalen Text. Daher müssen die Tag- und Koordinatensuche kodiert werden. Dies geschieht durch ein spezielles Schlüsselzeichen. Für die gesamte Anfrage wurde eine einfache Grammatik definiert. Abbildung 4.8 zeigt die Sprachdefinition einer korrekten Suchanfrage. Da die Suchanfrage schon während der Eingabe des Nutzers verarbeitet werden können soll, muss diese etwas relaxiert werden. So sollten fehlende schließende Klammern ergänzt, Operatoren mit fehlenden Operanden, sowie schließende Klammern ohne Gegenstück ignoriert werden. Die Suchanfrage *)(Wort1 ()* + *(Wort2* - ergibt somit *((Wort1 ()) + (Wort2))*. Für die Operatoren werden keine Präferenz definiert, sodass zusätzliche Operationen ohne Klammern wie ein Filter auf die vorhergehende Anfrage wirken (links-assoziativ).



Suchanfrage	:=	(Suchanfrage)
		Suchanfrage Operatorzeichen Suchanfrage
		Komplettierung
Komplettierung	:=	Textsuche
		Plugin-Suche
Textsuche	:=	Suchtext
Plugin-Suche	:=	$Plugin-Name(Plugin-Text)
Plugin-Name	:=	Text
Plugin-Text	:=	Text

Abbildung 4.8: Suchanfragegrammatik und zugehöriger Zustandsübergangsautomat (SA=Suchanfrage, OP=Operatorzeichen, TERM=Komplettierung)

5 Implementierung

5.1 Einleitung

In diesem Kapitel soll kurz auf die konkrete Implementierung der verwendeten Datenstrukturen eingegangen werden. Für diese Arbeit wurden verschiedene Programme und Bibliotheken entwickelt. Der Großteil der Entwicklung fand im Projekt *sserialize*, einer Bibliothek, die alle benötigten Datenstrukturen beinhaltet, statt. Um die Daten für die Nutzung auf dem Mobiltelefon bzw. Desktoprechner aufzubereiten, wurde *osmfind-create* entwickelt. Um die Daten auf dem Mobiltelefon zu nutzen, wurde OsmFind, eine Android-Applikation mit Kartendarstellung implementiert. Auf dem Desktoprechner kann OsmfindQt, welches auf dem qt-Toolkit aufbaut, genutzt werden. Desweiteren gibt es mit *osmfind* eine Konsolenanwendung, die sowohl auf dem Desktoprechner als auch auf dem Mobiltelefon verwendet werden kann.

5.2 sserialize

Da mobile Geräte weder über viel Speicher noch leistungsstarke Prozessoren verfügen, sollten die Datenstrukturen für die mobilen Geräte möglichst so implementiert werden, dass auch auf einzelne Teile, wie etwa ein Element einer Liste, ohne großen Speicher und Rechenaufwand zugegriffen werden kann. Hierzu wurde eine Serialisierungs-/Deserialisierungs-Bibliothek implementiert. Hierbei werden bei der Deserialisierung, abgesehen von der Wandlung von Network-Byteorder in Host-Byteorder, keine Daten dekodiert oder kopiert. Es werden nur diejenigen Teile geladen, die benötigt werden. Um ein effizientes Caching zu ermöglichen, wird die Datei mit den Daten mit mmap in den Arbeitsspeicher abgebildet. Dabei übernimmt das Betriebssystem die Verwaltung der Datei. Unter Linux wird die Datei in der Regel in 4 KB große Blöcke aufgeteilt. Wird nun ein Bereich der Datei angefordert, so werden all jene Blöcke geladen, in welchen der angeforderte Bereich liegt. Hierdurch kann sämtlicher zur Verfügung stehender Systemspeicher genutzt werden, ohne dass andere Anwendungen davon Nachteile haben, da das Betriebssystem, wenn mehr Arbeitsspeicher benötigt wird, ungenutzte Blöcke entfernt. Viele der Container besitzen überladene Streaming-Operatoren, sodass neue Datenstrukturen leicht serialisiert/deserialisiert werden können. Im Folgenden sollen diese kurz skizziert werden.

5.2.1 Externe Bibliotheken

Unicode ist ein Standard um Zeichen auf Zahlen, so genannte code points, abzubilden. Diese code points können sodann in vielfältiger Weise genutzt werden. Ein weiter verbreiteter Standard ist die utf8 Kodierung. Eine genaue Beschreibung des Unicode-Standards findet sich in [TA11]. Da die Bibliothek auf Zeichenketten mit utf8-Kodierung operieren können soll, wurde utf8cpp, eine C++-Bibliothek, von [Tri12] verwendet. Diese stellt Funktionen bereit, um über die Code-Points der Zeichenketten iterieren zu können. Um diakritische Zeichen zu entfernen, wurde [icu12] verwendet. Diese stellt umfangreiche Funktionen zum Umgang mit Unicode-Zeichenketten bereit.

5.2.2 Speicherzugriff

MmappedFile

Die MmappedFile-Klasse abstrahiert durch memory mapping den Dateizugriff. Sie stellt Methoden bereit, um Dateien zu erstellen, zu öffnen, in der Größe zu verändern sowie diese wieder zu schließen und bei Bedarf (automatisch) zu löschen.

ChunkedMmappedFile

Die ChunkedMmappedFile abstrahiert wie die MmappedFile-Klasse den Dateizugriff – mit dem Unterschied, dass die Datei nicht vollständig in den Arbeitsspeicher abgebildet wird. Stattdessen werden nur jene Teile abgebildet, die benötigt werden. Als Cache wird ein direkter Cache mit LFU als Ersetzungsstrategie verwendet.

UByteArrayAdapter

Der UByteArrayAdapter ist der Speicherunterbau für alle Container. Dieser abstrahiert den konkreten Speicherzugriff und stellt Serialisierungs- und Deserialisierungsfunktionen für die gängigen elementaren Datentypen bereit. Durch überladene Streaming-Operatoren können leicht neue zusammengesetzte Datenstrukturen entwickelt werden. Zum Zeitpunkt der Arbeit wurden Abstraktionsschichten für C-Arrays, RandomAccess-STL-Container, MmappedFile und ChunkedMmappedFile implementiert. Tabelle 5.1 gibt einen Überblick über die verwendete Kodierung.

Typ	Kodierung
Positive Zahlen konstanter Länge	In 8 Bit-Blöcken, von hochwertig zu niederwertig
Positive Zahlen variabler Länge	In 7 Bit-Blöcken, von niederwertig zu hochwertig, falls ein weiterer Block kodiert werden muss, wird das achte Bit der Zielzelle gesetzt
Negative Zahlen	Betrag der Zahl wird um 1 Bit nach links verschoben, wodurch das niederwertigste Bit zur Kodierung des Signums verwendet werden kann. Kodierung wie Positive Zahl
Zeichenketten	Kodierung in utf8. Zunächst die Länge als Positive Zahl variabler Länge, anschließend die Zeichenkette

Tabelle 5.1: Kodierung elementarer Datentypen in einen Byte-Array mit 8 Bit je Byte

5.2.3 Mengenartige

ItemIndex

Der ItemIndex ist eine Klasse, um eine Menge aus positiven Zahlen zu speichern. Die dahinterliegende konkrete Implementierung kann ausgetauscht werden. Als Speicherunterbau können beliebige RandomAcess-Container wie die STL-Deque/Vector oder statische Datenstrukturen, die in den folgenden Absätzen kurz erläutert werden, genutzt werden.

ItemIndex mit Regressionsgerade Oftmals lassen sich die Mengen, die als ItemIndex gespeichert werden, durch eine Regressionsgerade approximieren. Hierdurch müssen nur noch die Abweichungen der Elemente von der Regressionsgerade gespeichert werden. Durch einen zusätzlichen Offset werden die Abweichungen ins Positive verschoben, sodass keine negativen Zahlen kodiert werden müssen. Oft wird hierdurch eine Kompression erreicht, die unter der Entropie der Ausgangsdaten liegt.

Bytes	Datentyp	Bemerkung
1-4	varuint32	Kodiert die Bitweite der Elemente in den unteren 5 Bit, die Anzahl der Elemente in den restlichen 27 Bit
1-4	varint32	Ordinatenabschnitt
1-4	varuint32	Steigung der Geraden multipliziert mit der Anzahl der Elemente
1-4	varuint32	Offset, der von allen Zahlen abgezogen werden muss
*	CompactUintArray	Elementdaten

ItemIndex mit initialem Offset Da der ItemIndex die üblichen Mengenoperationen unterstützen muss, wird ein Unterbau benötigt, der möglichst einfach gehalten ist. Hierzu wurde der SimpleItemIndex implementiert, welcher lediglich das kleinste und größte Element sowie deren maximale Anzahl beim Erstellen benötigt. Innerhalb dieser Grenzen unterliegen die Elemente keinen Beschränkungen.

Bytes	Datentyp	Bemerkung
4	uint32	Kodiert die Byteweite der Elemente in den unteren 2 Bit, die Anzahl der Elemente in den restlichen 30 Bit
4	uint32	Offset für alle Elemente
*	CompactUintArray	Elementdaten

Bit-Vektor-basiert mit Laufweitenkodierung In Anlehnung an [WOS06] wurde eine Bit-Vektor-basierte Variante mit Laufweitenkodierung implementiert. Die Kodierung des Bit-Vektors findet dabei in 31-Bit-Schritten statt, sodass immer ein volles 32-Bit-Wort geladen wird. Im Gegensatz zu [WOS06] speichert das niederwertigste Bit, ob es sich um ein laufweitenkodiertes Wort handelt. Hierdurch muss die CPU-Architektur keine konstanten 32-Bit-Operationen unterstützen, da lediglich die 2 niederwertigsten Bits benötigt werden, um ein laufweitenkodiertes Wort zu dekodieren. Die Anzahl der kodierten Worte lässt sich sodann durch zweifaches nach rechts schieben erhalten.

Bytes	Datentyp	Bemerkung
4	uint32	Kodiert die Datengröße
4	uint32	Kodiert die Anzahl der Elemente
*	uint32	Kodiert den Bit-Vektor

Bit	Bemerkung
0	Kodiert den Typ (Laufweitenkodierung oder nicht)
1	Kodiert den Wert der Laufweitenkodierung
2-31	Kodiert die Anzahl der laufweitenkodierten Worte

Differenz-Kodiert Der Differenz-kodierte ItemIndex kodiert die Daten durch eine Differenz-Kodierung zum jeweils vorhergehenden Element. Auch dieser ItemIndex besitzt daher keinen wahlfreien Zugriff.

Bytes	Datentyp	Bemerkung
4	uint32	Kodiert die Datengröße
4	uint32	Kodiert die Anzahl der Elemente
*	vluint32	Kodiert die Differenzen

Static::Set

Der Set-Container speichert eine Menge aus Elementen in einem Array in sortierter Reihenfolge. Elemente können so mit einer Binärsuche gefunden werden.

Bytes	Datentyp	Bemerkung
1	uint8	Versionsnummer
4	uint32	Datenlänge
*	*	Serialisierte Daten in sortierter Reihenfolge
*	ItemIndex	Index mit Offsets zu den einzelnen Dateneinträgen

5.2.4 Array-artige

Vector

Um Arrays aus beliebigen Objekten zu serialisieren, wurde die Static::Vector-Klasse implementiert. Diese bietet einen einfachen wahlfreien Zugriff auf die serialisierten Daten. Darüberhinaus existiert eine Klasse, um eine Static::Vector-Instanz sukzessive aufzubauen.

Bytes	Datentyp	Bemerkung
1	uint8	Versionsnummer
4	uint32	Datenlänge
*	*	Serialisierte Daten
*	ItemIndex	Index mit Offsets zu den einzelnen Dateneinträgen

CompactUintArray

Der CompactUintArray ist eine einfache Wrapperklasse um Zahlen mit unterschiedlichen Bitweiten in einem unsigned byte Array zu speichern. Diese Klasse wird in vielen weiteren Datenstrukturen zur Kodierung verwendet.

MultiVarBitArray

Der MultiVarBitArray ist ähnlich wie der CompactUintArray. Der MultiVarBitArray ermöglicht darüber hinaus, je Position mehrere Zahlen unterschiedlicher Bitweite zu speichern. Die Bitweiten der einzelnen Zahlen ist hierbei konstant.

Map

Mit der Map-Datenstruktur können Schlüssel auf Werte abgebildet werden. Dabei werden die Schlüssel in sortierter Reihenfolge abgespeichert, sodass eine Binärsuche verwendet werden kann.

Bytes	Datentyp	Bemerkung
1	uint8	Versionsnummer
4	uint32	Datenlänge
*	*	Paare aus Datentyp-1 und Datentyp-2, sortiert abgelegt nach Datentyp-1
*	ItemIndex	Index mit Offsets zu den einzelnen Dateneinträgen

5.2.5 Generalisierter Suffix Trie

Knotentypen

Für die Knoten wurde eine einheitliche Schnittstelle definiert. Hierdurch lässt sich der konkrete Unterbau und somit die genaue Serialisierung/Deserialisierung leicht ändern. Zu Vergleichszwecken wurden 2 unterschiedliche Knotentypen implementiert.

SimpleTrieNode Der einfachste Knotentyp speichert alle nötigen Informationen in elementaren Datentypen ab. Entsprechend hoch ist der Speicherbedarf.

Bytes	Datentyp	Bemerkung
2	uint16	Anzahl der Kindknoten
1	uint8	Byte-Breite der Knotenzeichen der Kindknotenzeiger
1	uint8	Indextyp
1	uint8	Zeichenkettenlänge
*	uint8	Knotenzeichenkette
*	uint16	Zeichen der Kindknotenzeiger
*	uint32	Zeichenkettenlänge
1	uint8	Kindknotenzeiger
4*4	uint32	ItemIndex-Zeiger

CompactTrieNode Ein komplexeres Speicherschema wurde mit dem CompactTrieNode realisiert.

Bits	Datentyp	Bemerkung
1	bool	Breite der Zeichen der Kindknotenzeiger
1	bool	Gibt an, ob es sich um einen merge Item-Index handelt
1	bool	Gibt an, ob ein weiteres Kindknotenanzahl-Byte nach dem Header folgt
5	uint	Anzahl der Bits für die ItemIndex-Zeiger
4	bool	Jedes Bit gibt an, welche ItemIndex-Zeiger vorhanden sind
4	uint	Anzahl der Kindknotenzeiger
8	uint8	Anzahl der Kindknotenzeiger (optional, gibt die unteren 8 Bit an)
8	uint8	Knotenzeichenkettenlänge
*	uint8	Knotenzeichenkette
8-16	uint8/16	Zeichen der Kindknotenzeiger
32	uint32	Zeiger auf den ersten Kindknoten
*	uint16	Zeiger auf die restlichen Kindknoten als Offset
*	CompactUintArray	ItemIndex-Zeiger

Top-Down-Serialisierung

Der Baum kann auf zwei unterschiedliche Varianten serialisiert werden. Im Top-Down-Verfahren wird der Baum in Level-Order von der Wurzel zu den Blättern serialisiert. Kindknotenzeiger müssen dabei nachträglich korrigiert werden. Da hierbei die Größe der Kindknotenzeiger bei der Serialisierung eines Knotens noch nicht bekannt ist, muss entsprechend viel Platz gelassen werden.

Bottom-Up-Serialisierung

Alternativ kann der Baum in Level-Order von den Blättern zu der Wurzel serialisiert werden. Die Kindknotenzeiger sind so bei der Serialisierung eines Knotens bereits bekannt. Im Gegensatz zur Top-Down-Serialisierung sind die Kindknotenzeiger relative Offsets vom aktuellen Knoten.

FlatGST

Der FlatGST implementiert eine Suffixarray-ähnliche Struktur. Hierbei werden die Knoten implizit abgespeichert. Knotenzeichenketten, Kindknotenzeiger und deren Zeichen entfallen. Stattdessen wird der Baum in In-Order-Reihenfolge in einem Array abgespeichert. Jeder Knoten kann hierbei eindeutig über seinen Pfad im Baum identifiziert werden. Da die Kindknoten sortiert vorliegen, liegen die gesamten Knoten im Array sortiert nach dem Pfad vor. Ein Knoten kann somit mit einer Binärsuche gefunden werden. Statt für jeden Knoten seinen vollen Pfad zu speichern, kann eine Referenz auf eine Zeichenkette, die diesen Pfad enthält, gespeichert werden. Hierdurch muss je Knoten ein Zeiger auf diese Zeichenkette, die Startposition innerhalb dieser und die Länge des Pfades gespeichert werden.

Bytes	Datentyp	Bemerkung
1	uint8	Versionsnummer
1	uint8	Gibt an, welche Anfragen unterstützt werden
*	StringTable	Zeichenkettentabelle
*	MultiVarBitArray	Knotenzeichenkettenzeiger
*	Vector	Array aus ItemIndex-Zeigern für jeden Knoten

5.2.6 Datenbanken

StringsItemDB

Die StringsItemDB dient dazu, Elemente mit mehreren Zeichenketten effizient abzuspeichern. Statt jede Zeichenkette in jedem Element zu speichern, erhalten die Elemente eine Referenz auf die Zeichenkette. Hierdurch wird jede Zeichenkette nur einmal abgespeichert. Die Zeichenketteninformationen der Elemente werden getrennt von diesen am Stück abgespeichert. Hierdurch können die Zeichenketten der Elemente effizient durchsucht werden.

Bytes	Datentyp	Bemerkung
1	uint8	Versionsnummer
*	Vector	Array aus Zeichenkettenzeigern der Elemente
*	Vector	Array aus Elementinformationen

GeoStringsItemDB

Die GeoStringsItemDB leitet sich von der StringsItemDB ab. Zusätzlich können für jedes Element geometrische Informationen abgelegt werden. Diese Daten werden ebenfalls am Stück abgespeichert, wodurch die Elemente Cache-effizient durchsucht werden können.

Bytes	Datentyp	Bemerkung
1	uint8	Versionsnummer
*	Vector	Array aus geometrischen Objekten der Elemente
*	StringsItemDB	Zeichenketteninformationen und Elementinformationen

5.2.7 TagStore

Der TagStore ähnelt einer Ordnerstruktur. Knoten besitzen einen Namen, einen ItemIndex-Zeiger sowie Kinder.

Bytes	Datentyp	Bemerkung
2	uint16	Elterknotenreferenz
2	uint16	Position im Elterknoten
4	uint32	ItemIndex-Zeiger
*	Map	Zuordnung zwischen Kindknotenzeichenkette und Kindknotenreferenz
*	Vector	Array aus ItemIndex-Zeigern für jeden Knoten

Bytes	Datentyp	Bemerkung
1	uint8	Versionsnummer
4	uint32	Größe
*	ItemIndex	Offsets zu den Kindknoten
*	TagStore::Node	Knoten des Baumes

5.2.8 GeoGrid

Das GeoGrid ist ein einfaches 2D-Raster, welches eine Abbildung von Gitterzellen zu beliebigen Daten definiert.

Bytes	Datentyp	Bemerkung
1	uint8	Versionsnummer
*	GeoPoint	Grenzrechteck
*	GeoPoint	Grenzrechteck
4	uint32	Anzahl der Zellen in Breitengradrichtung
4	uint32	Anzahl der Zellen in Längengradrichtung
*	StorageType	Datentyp, der den Inhalt der Zellen enthält

ItemGeoGrid

Das ItemGeoGrid basiert auf dem GeoGrid, wobei jede Zelle einen Zeiger auf den ItemIndex, welcher alle Elemente enthält, die in dieser Zelle enthalten sind, beinhaltet.

5.3 osmfind

osmfind ist das Dachprojekt für alle Unterprogramme, die die OpenStreetMap-Daten verarbeiten. Zunächst müssen die Rohdaten verarbeitet werden. Für diesen Schritt wurde das Programm osmfind-create implementiert. Die neuen Daten können nun von weiteren Programmen genutzt werden. Zur einfachen Nutzung wurde eine Bibliothek mit allen relevanten Funktionen erstellt. Für Java-Programme gibt es JNI-basierte Wrapperklassen.

5.3.1 osmfind-create

osmfind-create ist das Hauptprogramm um die nötigen Komplettierungsdaten zu erstellen. Es handelt sich dabei um ein Konsolenprogramm. Mit Kommandozeilenoptionen lassen sich die Optionen für das Erstellen bestimmen. Tabelle 5.2 erläutert die einzelnen Optionen.

5.3.2 libosmfind

libosmfind ist eine Bibliothek zur Nutzung der Daten, die von osmfind-create erstellt werden. Sie stellt Funktionen und Klassen bereit, die den Zugriff auf diese Daten abstrahieren. Im Kern besteht sie aus Template-Instanzen von Klassen von sserialize mit speziellen Klassen, um die Daten für OpenStreetMap abspeichern zu können.

5.3.3 osmfind

osmfind ist ein Konsolenprogramm, mit welchem die OpenStreetMap-Komplettierungsdaten genutzt werden können. Das Programm bietet die Möglichkeit, verschiedene Statistiken auszugeben, eine oder mehrere Komplettierungen durchzuführen und einen interaktiven Modus.

5.3.4 osmfindqt

osmfindqt ist eine Desktopapplikation auf Basis des QT-Projekts[qt-], um auf einem Rechner die Komplettierungsdaten komfortabel Nutzen können. Es können zur Laufzeit die Komplettierer ausgewählt sowie die Tag-Suche genutzt werden. Im linken unteren Feld werden darüberhinaus einige Statistiken angezeigt sowie Zeichen, die eine valide Komplettierung darstellen. Abbildung 5.1 zeigt die Oberfläche mit einer komplexeren Suchanfrage.

-c	Erzeugt Komplettierungsdaten, die die Groß-/Kleinschreibung beachten
-s	Erzeugt Komplettierungsdaten, die eine Suffixsuche ermöglichen
-adi	Fügt zusätzliche jede Zeichenkette ohne ihre diakritische Zeichen ein
-sd	Definiert im Falle einer Suffix-Suche die Trennzeichen, um die Suffixzeichenketten zu erstellen
-d	Erstellt einen direkten Index
-w	Setzt eine feste Bitweite für den ItemIndex
-it Typ	Definiert den ItemIndex-Typ (Regressionsgerade=rline, Bit-Vektor=wah, Differenzkodierung=de)
-o	Schaltet die Approximierung mit einer Regressionsgerade aus
-nf Zahl	Zahl definiert eine Ebene im Baum, die keinen Präfix-/Teilzeichenketten-ItemIndex bekommen soll
-nfr Zahl Zahl	Definiert ein Intervall von Ebenen, die keinen Präfix-/Teilzeichenketten-ItemIndex Bekommen sollen
-nfpi Zahl	Definiert eine Obergrenze für die Anzahl der Vergleiche, um den Präfixzeichenketten-ItemIndex eines Knotens aus seinen Kindern zu erstellen, unterhalb derer kein voller Präfixzeichenketten-ItemIndex erstellt wird
-nfsi Zahl	Definiert eine Obergrenze für die Anzahl der Vergleiche, um den Teilzeichenketten-ItemIndex eines Knotens aus seinen Kindern zu erstellen, unterhalb derer kein voller Teilzeichenketten-ItemIndex erstellt wird
-nt Knotentyp	Setzt den Knotentyp für die Serialisierung
-b	Serialisiert den Baum von unten nach oben
-p lat lon	Definiert die Anzahl der Zellen, die beim Schneiden mit den Grenzpolygonen genutzt werden sollen
-a	Versucht so viel Arbeitsspeicher wie möglich während der Serialisierung frei zu geben
-nid Zahl	Definiert eine Untergrenze für den FlatGST, unterhalb derer kein Zeichenkettenreferenz-ItemIndex, sondern ein Element-ItemIndex erstellt wird
-mslid	Definiert die minimale Zeichenkettenlänge, um im FlatGST einen Element-ItemIndex zu erstellen
-ifs Zahl	Setzt die Größe der temporären Datei für die ItemIndexFactory
-tempdir Pfad	Setzt den Pfad zu den temporären Dateien
–no-merge-index	Schaltet die Nutzung der Teilmengenrelationen zwischen den einzelnen ItemIndex-Indices eines Knotens aus
–trie	Erzeugt einen serialisierten Baum
–fgst	Erzeugt einen FlatGST
–fgststrids	Erzeugt einen FlatGST mit Zeichenkettenindex
–create-grid lat lon	Erzeugt ein Raster für die Koordinatenkomplettierung
–extensive-checking	Überprüft die serialisierten Daten auf Korrektheit
-o Pfad	Pfad zur Ausgabedatei
-oa	Automatische Benennung der Ausgabedatei in Abhängigkeit der gewählten Optionen

Tabelle 5.2: Kommandozeilenoptionen für osmfind-create

Option	Beschreibung
-s all,idxstore,completer,db,geo,tag	Gibt Statistiken aus
-c Zahl	Selektiert den Zeichenkettenkomplettierer
-g Zahl	Selektiert den Koordinatenkomplettierer
-p Anzahl	Selektiert die Nutzung der Iterator-basierten Komplettierung und liest Zahl Elemente
-m	Definiert eine Zeichenkette, die komplettiert werden soll
-m	Definiert eine Datei, aus welcher Zeichenketten zur Komplettierung geladen werden sollen
-mdb Anzahl	Definiert die Anzahl von Komplettierungszeichenketten, die aus der Datenbank erstellt werden sollen
–simulate	Komplettierungszeichenketten werden zur Simulation einer menschlichen Eingabe genutzt
–benchmark Zahl1 Zahl2 Zahl3	Führt einen Leistungstest durch. Zahl3 gibt die Anzahl der Testläufe an, Zahl2 die Anzahl der Zeichenketten, die genutzt werden sollen um die Komplettierungsanfrage zu erstellen, Zahl3 die minimale Länge der einzelnen Testzeichenketten

Tabelle 5.3: Kommandozeilenoptionen von osmfind

5.3.5 OsmFind

OsmFind ist eine Androidanwendung, die die Nutzung der OpenStreetMap-Komplettierungsdaten ermöglicht. Für eine gegebene Komplettierungszeichenkette wird eine Liste der passenden Elemente angezeigt. Darüberhinaus können diese auf einer Karte visualisiert werden. In dieser Karte können sowohl Koordinatenkomplettierungen als auch Koordinateneinschränkungen für die Ergebnismengen definiert werden. Die Benutzeroberfläche wurde in Java geschrieben und benutzt die Java-Schnittstelle von libosmfind. Für die Darstellung der Karte wurde mapsforge von [map], eine Bibliothek zur Darstellung von OpenStreetMap-Karten, verwendet. Abbildungen 5.2, 5.3 und 5.4 zeigen die verschiedenen Nutzungsmöglichkeiten.

5.3.6 Nutzung

Im folgenden wird ein kurzes Codebeispiel zur Nutzung der Komplettierung gegeben:

```
1  #include <libosmfind/StaticOsmCompleter.h>
2
3  void complete() {
4
5  std::string inFileName = "prefix/der/komplettierungs/daten";
6
7  //Relevante Daten setzen
8  Static::OsmCompleter osmCompletion;
9  osmCompletion.rcInc();
10 osmCompletion.setIndexFile(inFileName+".index");
11 osmCompletion.setStringTableFile(inFileName+".strings");
12 osmCompletion.setMetaDataFile(inFileName+".data");
13 osmCompletion.setTagCompleterFile(inFileName+".tagstore");
14
15 //Spezielle schnelle Komplettierer setzen
16 osmCompletion.addStringCompleterFile(inFileName+ ".trie");
17 osmCompletion.addStringCompleterFile(inFileName+ ".flatgst");
18 osmCompletion.addStringCompleterFile(inFileName+ ".fgstrids");
19 osmCompletion.addGeoCompleterFile(inFileName + ".grid");
20
21 //aktivieren
22 if (!osmCompletion.energize())
23    return;
24
25 //aktiven Zeichenkettenkomplettieren setzen
26 if (!osmCompletion.setCompleter(selectedCompleter))
27    return;
28
29 //aktiven Geokomplettierer setzen
30 if (!osmCompletionTrie.setGeoCompleter(selectedGeoCompleter))
31       return;
32
33 //Iterator-Schnittstelle (führt Mengenoperationen dynamisch aus)
34 Static::OsmItemSetIterator itemSetIterator = osmCompletion.partialComplete("initial
       string");
35
36 //Direktzugriff-Schnittstelle (führt Mengenoperationen vollständig aus)
37 Static::OsmItemSet itemSet = osmCompletion.complete("initial string");
38
39 //Mehrere Möglichkeiten Elemente zu analysieren
40 Static::OsmItemDBItem item = *itemSetIterator;
41 Static::OsmItemDBItem item = itemSetIterator.at(1);
42 Static::OsmItemDBItem item = itemSet.at(0);
43
44 //Die Komplettierungszeichenkette aktualisieren
45 itemSetIterator.update("updated string");
46 itemSet.update("updated string");
47
48 }
```

5.4 Verbesserungsmöglichkeiten

Je nach Anwendungsszenario sind unterschiedliche Verbesserungsmöglichkeiten interessant.

Nebenläufigkeit Für den Einsatz als Serversoftware, um z.B. Anfragen einer Webapplikation zu erfüllen, sollten sämtliche Funktionen Nebenläufigkeit ermöglichen. Hierzu müssen sämtliche Strukturen, die von mehreren Threads geteilt werden, gegen gegenseitige Beeinflussung gesichert werden. Dies sind vor allem die Referenzzähler sowie sämtliche Cachedatenstrukturen. Desweiteren muss das Anlegen von Cachedateien über eine zentrale Anlaufstelle geschehen, um zu verhindern, dass 2 Threads die gleichen Dateien anlegen. Eine andere Möglichkeit wäre es, die Dateinamen mit dem konkreten Thread zu verknüpfen.

Zustandslosigkeit Um möglichst wenige Informationen zu speichern sollte die Schnittstelle zustandslos verwendet sein. Dies kann z.B. erfolgen, indem Aktualisierungen der Komplettierung nicht erfolgen und stattdessen die Anfrage komplett neu erfolgt. Hierzu könnten einzelne Unterbäume der Mengenoperationsbäume extra gecacht werden. Bei der Beantwortung einer Anfrage könnten nun die vorberechneten Teilbäume wiederverwendet werden.

Strukturierte Suche Interessant wäre auch eine strukturierte Suche, sodass z. B. nur der *name*-Tag durchsucht werden könnten. Dies könnte leicht durch Einführung eines neuen Plugin-Komplettierers gelöst werden. So könnte der *name*-Tag z. B. durch **$NAME[***Suchzeichenkette***]** durchsucht werden.

Kompression Wie sich in Kapitel 6 zeigen wird, kann eine Kompression der Daten eine weitere Verringerung des Speicherplatzbedarfes für bestimmte ItemIndex-Typen erreichen. Auch wäre es sinnvoll die Verwendung von komprimierten Textindices abzuklären. Beide Erweiterungen lassen sich jeweils als Unterbau des ItemIndex respektive eines StringCompleters realisieren.

Internationalisierung Da OpenStreetMap ein internationales Projekt ist, sollte nach Möglichkeit in verschiedenen Sprachen gesucht werden können, um so auch landesspezifische Namen erfassen zu können.

Dynamische Indices Die bisherige Implementierung der Vereinigung vieler Indices nutzt einen Baum-basierten merge-Algorithmus, was im Falle von n Indices mit m Elementen im schlechtesten Fall $O\left(n \cdot \log_2\left(n\right) \cdot m\right)$ Speicherzugriffe benötigt. Dies könnte mit Hilfe eines Bit-Vektors ohne Laufweitenkodierung auf $O\left(n \cdot m\right)$ verringert werden.

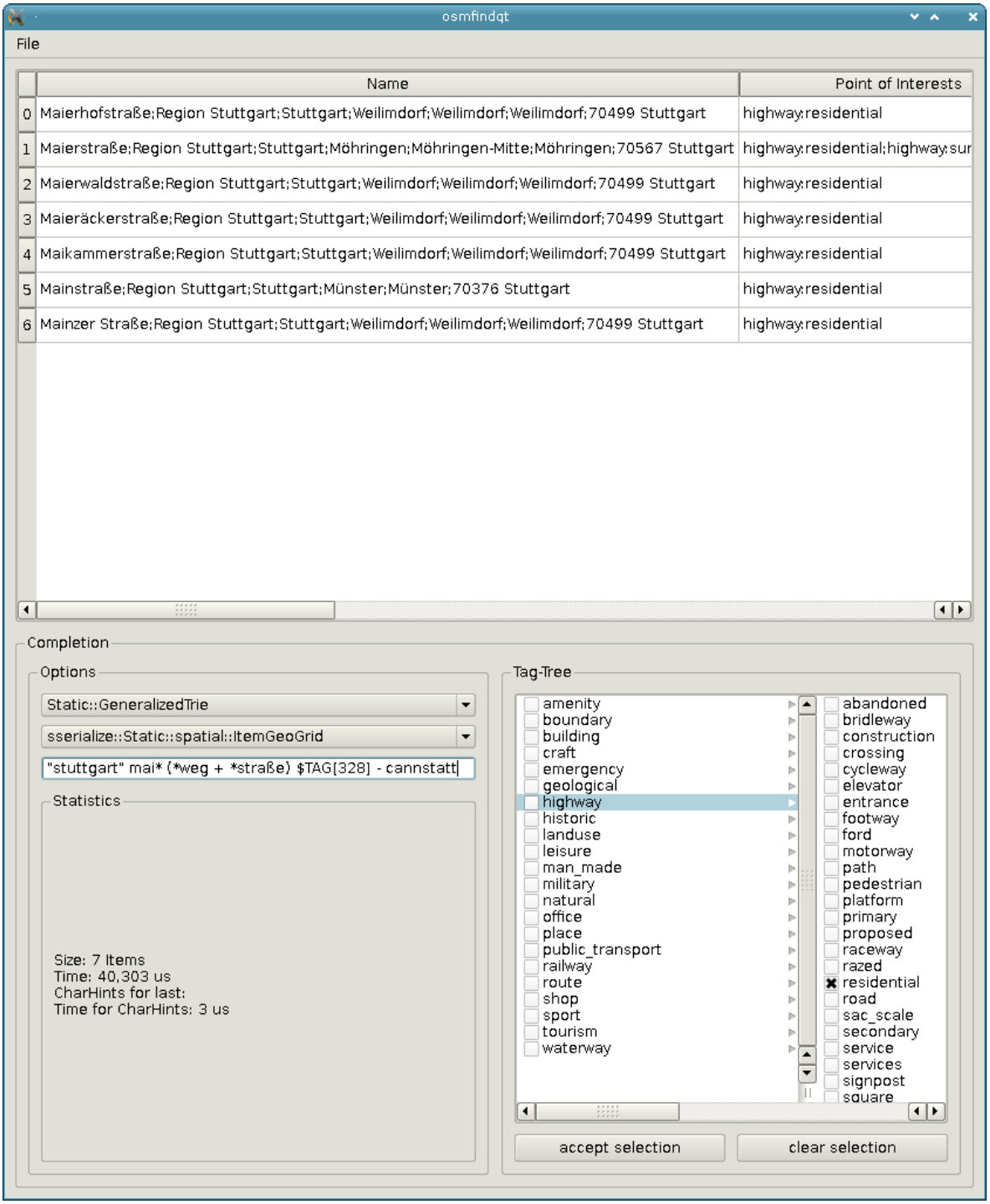


Abbildung 5.1: Qt-Desktopapplikation osmfindqt mit einer komplexeren Suchanfrage

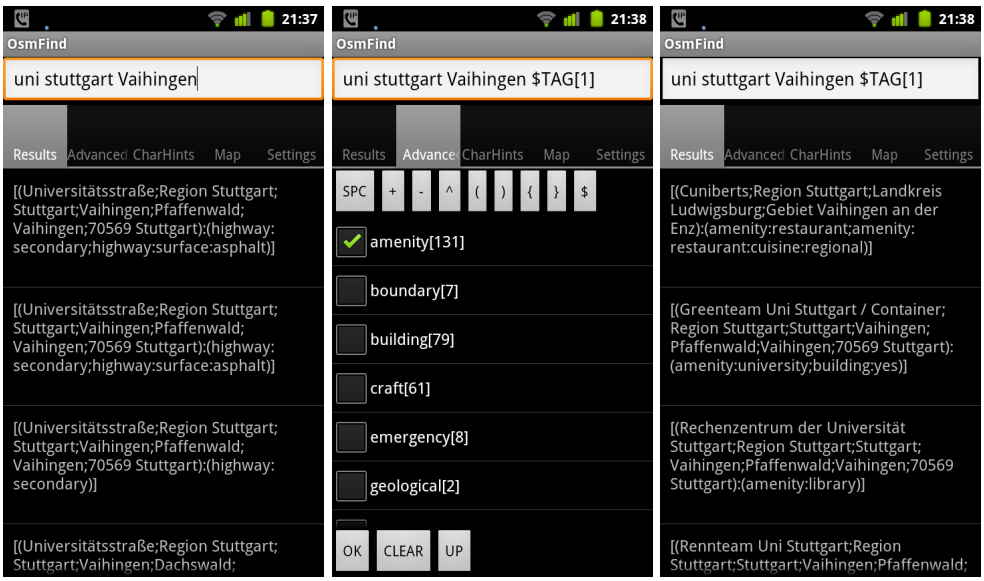


Abbildung 5.2: Android-Applikation OsmFind: Links: Komplettierung von Zeichenketten mit Schnitt. Mitte: Auswahlfenster der Tags, Rechts: Änderung durch die Auswahl der Tag-Kategorie “amenity”

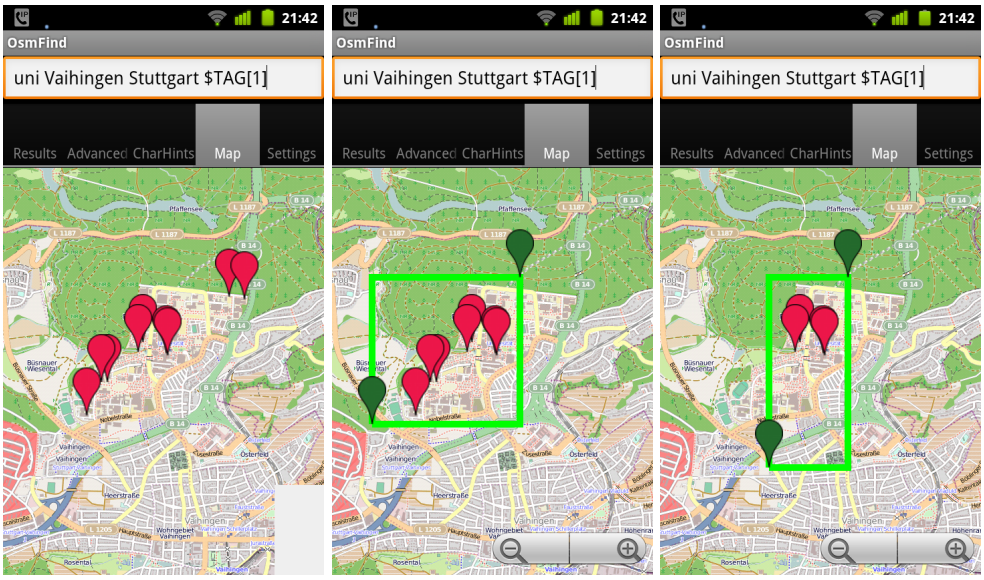


Abbildung 5.3: Android-Applikation OsmFind: Links: Karte zur Komplettierung mit Markern für die gefunden Elemente, Mitte: Einschänkung der Elemente auf Basis ihrer Koordinaten, Rechts: Änderung der Koordinateneinschränkung

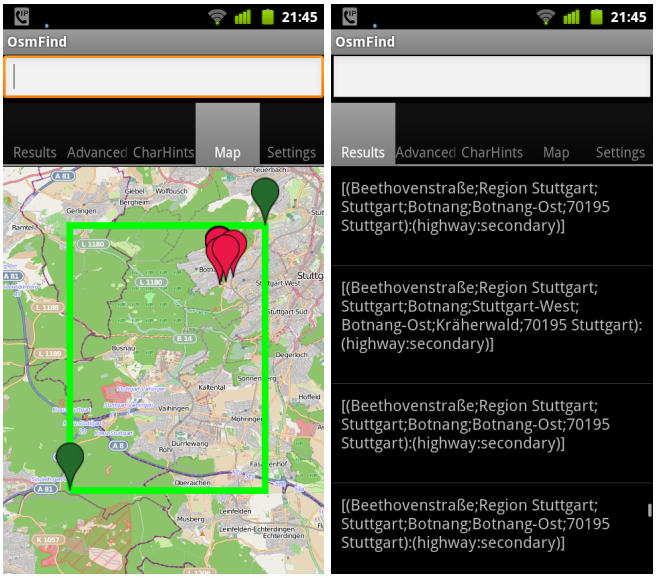


Abbildung 5.4: Android-Applikation OsmFind: Links: Auswahl von Element nur nach ihren Koordinaten, Rechts: Liste der so erhaltenen Elementen

6 Ergebnisse

In diesem Kapitel sollen abschließend einige Statistiken vorgestellt werden. Hierzu wurden verschiedene Datensätze, die von der Geofabrik erstellt wurden, analysiert. Zunächst wurden die Grenzpolygone mit Hilfe des Programms boundaries.pl extrahiert. Anschließend wurden Komplettierungsdateien mit unterschiedlichen Einstellungen erstellt, welche sodann mit synthetisch erstellten Komplettierungsanfragen getestet wurden. Die Leistungsdaten des in der Vorverarbeitung und bei Leistungsanalysen genutzten Desktoprechner sind in Tabelle 6.2 wiedergegeben. Zur Messung der Leistung auf dem Mobiltelefon kam ein Motorola Defy zum Einsatz, dessen Kenndaten in Tabelle 6.3 zu finden sind.

Mengen-Operationen Um die Geschwindigkeit der Mengenoperationen zu messen, wurden zwei unterschiedliche Verfahren angewandt. Um die reine Durchführungszeiten der Mengenoperationen zu testen, wurden zufällig erzeugte Mengen genutzt. Der Mengenoperationsbaum wurde getestet, indem Komplettierungsanfragen simuliert wurden, um so die Geschwindigkeitsverbesserung durch die Aktualisierung zu erfassen, als auch die Auswirkungen verschiedener ItemIndex-Speicherschemata. Abbildung 6.2 zeigt den Algorithmus zur Erzeugung der Komplettierungsanfragen. Für die Messungen wurden 10 Komplettierungsanfragen erstellt. Aus jedem Element wurden hierfür 5 Teilzeichenketten mit einer Mindestlänge von 3 Zeichen für die Komplettierung genutzt.

Komplettierung Um verschiedene Komplettierer und ItemIndex-Speicherschemata zu vergleichen, fanden für die Messungen der Komplettierungszeiten keine Mengenoperationen statt. Stattdessen wurden mit Hilfe des in Abbildung 6.1 beschriebenen Algorithmus eine gewisse Anzahl unterschiedlicher Zeichenketten erzeugt. Für jede dieser Zeichenketten wurde nun eine Simulation einer Eingabe durchgeführt und dabei die Gesamtzeit sowie die Zeit jeder einzelnen Komplettierung gemessen. Caching-Effekte, die z.B. beim GST auftreten können, wurden dabei nicht verhindert, da ein Nutzer der Suchfunktion bei der Eingabe ebenfalls von jenen Caching-Effekten profitieren würde. Für die Messungen wurden 100 Komplettierungsanfragen erstellt mit einer Mindestlänge von 3 Zeichen.

Abkürzungen Zur kürzeren Darstellung werden in diesem Kapitel einige Abkürzungen verwendet, die größtenteils mit den Optionen von *osmfind-create* übereinstimmen und in Tabelle 6.1 aufgeführt sind.

Abkürzung	Kontext	Bedeutung
voller	ItemIndex Option	Teilmengenrelationen werden nicht zur Speicherplatzverringerung genutzt
merge	ItemIndex Option	Teilmengenrelationen werden zur Speicherplatzverringerung genutzt
nfr=(Zahl, Zahl)	ItemIndex Option	Gibt einen Bereich von Ebenen an, für die kein Präfix- und Teilzeichenkettenindex erstellt wird
nfpi=Zahl	ItemIndex Option	Definiert eine Obergrenze für die Anzahl der Vergleiche um den Präfixzeichenketten-ItemIndex eines Knotens aus seinen Kindern zu erstellen, unterhalb derer kein voller Präfixzeichenketten-ItemIndex erstellt wird
nfsi=Zahl	ItemIndex Option	Definiert eine Obergrenze für die Anzahl der Vergleiche um den Teilzeichenketten-ItemIndex eines Knotens aus seinen Kindern zu erstellen, unterhalb derer kein voller Teilzeichenketten-ItemIndex erstellt wird
nid=Zahl	FlatGST Option	Definiert eine Untergrenze für den FlatGST, unterhalb derer kein Zeichenkettenreferenz-ItemIndex, sondern ein Element-ItemIndex erstellt wird
msl=Zahl	FlatGST Option	Definiert die minimale Zeichenkettenlänge, um im FlatGST einen Element-ItemIndex zu erstellen
-s	Trie Option	Erstellt einen Generalisierten Suffix-Trie
-s -adi	Trie Option	Erstellt einen Generalisierten Suffix-Trie und fügt zusätzliche jede Zeichenkette ohne ihre diakritischen Zeichen ein
-sd	Trie Option	Definiert im Falle einer Suffix-Suche die Trennzeichen, um die Suffixzeichenketten zu erstellen. Folgende Zeichen wurden hierfür verwendet: *,;.:-_#*$/?()[]{}+ |<>*

Tabelle 6.1: Abkürzungsverzeichnis der Ergebnisse

CPU	Intel Core i7 930@3,8 GHz
Arbeitsspeicher	12 GB Triple-Channel 1333MHz
Festplatten	2 WD6400AAKS Raid1
Dateisystem	ext4
Betriebssystem	Linux 3.4.5-hardened
Compiler	gcc (Gentoo Hardened 4.6.3 p1.6, pie-0.5.2) 4.6.3
Compileroptionen	-nopie -fno-stack-protector -O3 -march=native

Tabelle 6.2: Leistungsdaten und Compileroptionen des Desktoprechners

Mobiltelefon	Motorola Defy mit CyanogenMod 7.1
CPU	OMAP3610-800@1000MHz
Arbeitsspeicher	512 MB
SD-Karte	Samsung Essential Class 10 microSDHC 32GB
Compiler	android-ndk8b mit gcc-4.6
Compileroptionen	-march=armv7-a -mfloat-abi=softfp -mfpu=neon

Tabelle 6.3: Leistungsdaten und Compileroptionen des Mobiltelefons

6.1 Mengen-Operationen

Abbildungen 6.1 und 6.2 zeigen für verschiedene ItemIndex-Speicherschemata die Zeit in Abhängigkeit von der Größe, um zwei Mengen zu vereinen. Am schnellsten war hierbei der ItemIndex mit einem std::vector als Unterbau, dicht gefolgt vom einfachen ItemIndex, welcher vor allem durch die Kodierung der Elemente in Netzwerk-Byte-Order langsamer war. Von den Speicherschemata die zur Speicherung dienen, stach vor allem der Bit-Vektor-basierte Index hervor. Allen gemeinsam war die klare lineare Abhängigkeit von der Größe der Eingabemenge.

6.2 Datensätze

In der Vorverarbeitung müssen zunächst die Daten in den Hauptspeicher eingelesen und aufbereitet werden. Die Geschwindigkeit der Festplatten stellte dabei auf dem Testsystem keinen Flaschenhals dar, da die benötigte Lesegeschwindigkeit weit unter der möglichen Geschwindigkeit der Festplatten lag. Tabelle 6.4 gibt einen Überblick über die Laufzeit und den Arbeitsspeicherbedarf einiger Datensätze. Tabelle 6.5 zeigt die Verteilung der Dateigrößen für verschiedene Datensätze. Der ItemIndexStore benötigte hierbei gegenüber den restlichen Daten oft den zehnfachen Platz. Die Dateigröße des TagStore hingegen ist vernachlässigbar gering. Tabelle 6.6 zeigt einige Kenndaten wie die Anzahl der unterschiedlichen Zeichenketten für diverse Datensätze. Im Schnitt wird dabei jede Zeichenkette in 3 Elementen genutzt. Tabelle 6.7 zeigt die Kompressibilität für verschiedene Datensätze. Um die Kompressibilität zu testen, wurde lrzip von [Kov] und lzop von [Obe] verwendet. Lrzip

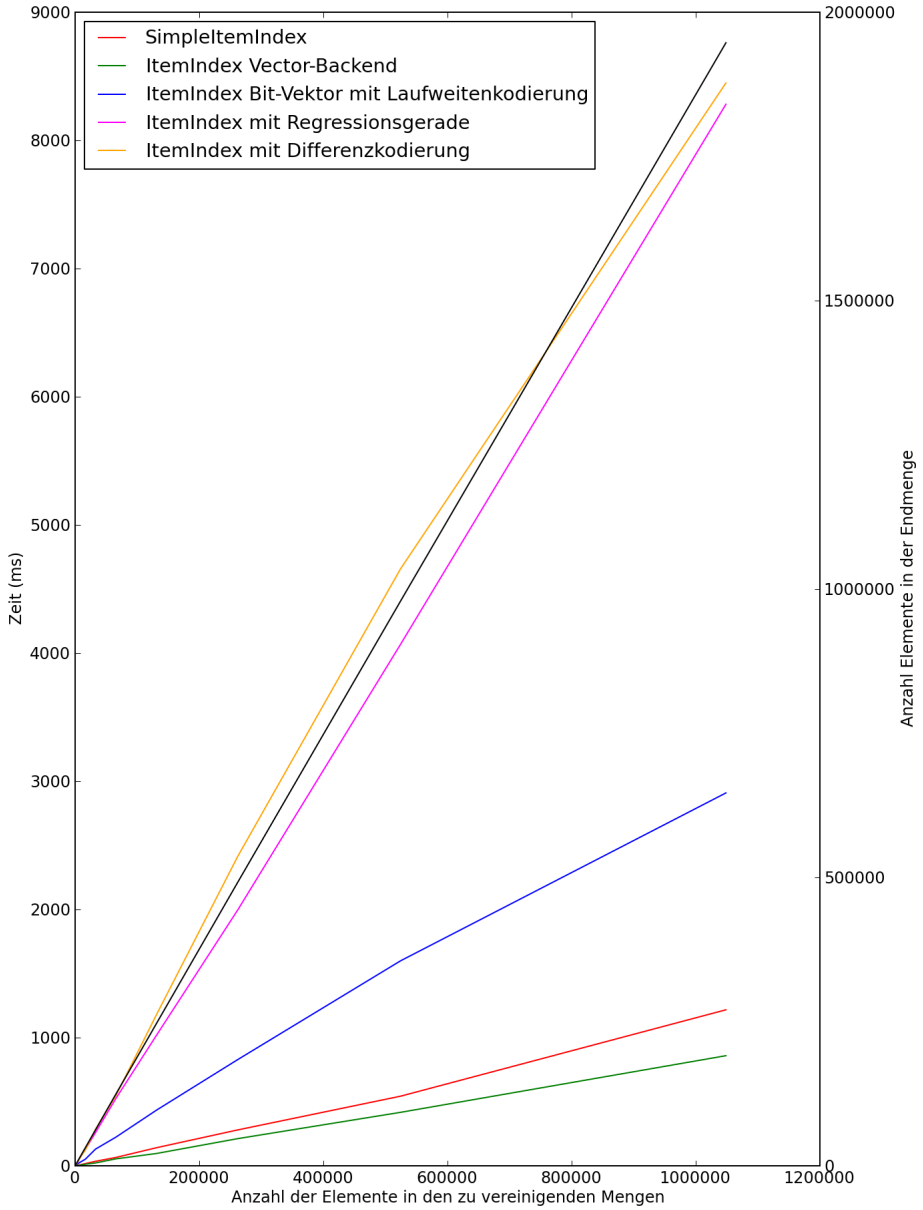


Abbildung 6.1: Vereinigung zweier Mengen für unterschiedliche ItemIndex Speicherschemata auf dem Mobiltelefon, Größe der Ergebnismenge in Schwarz

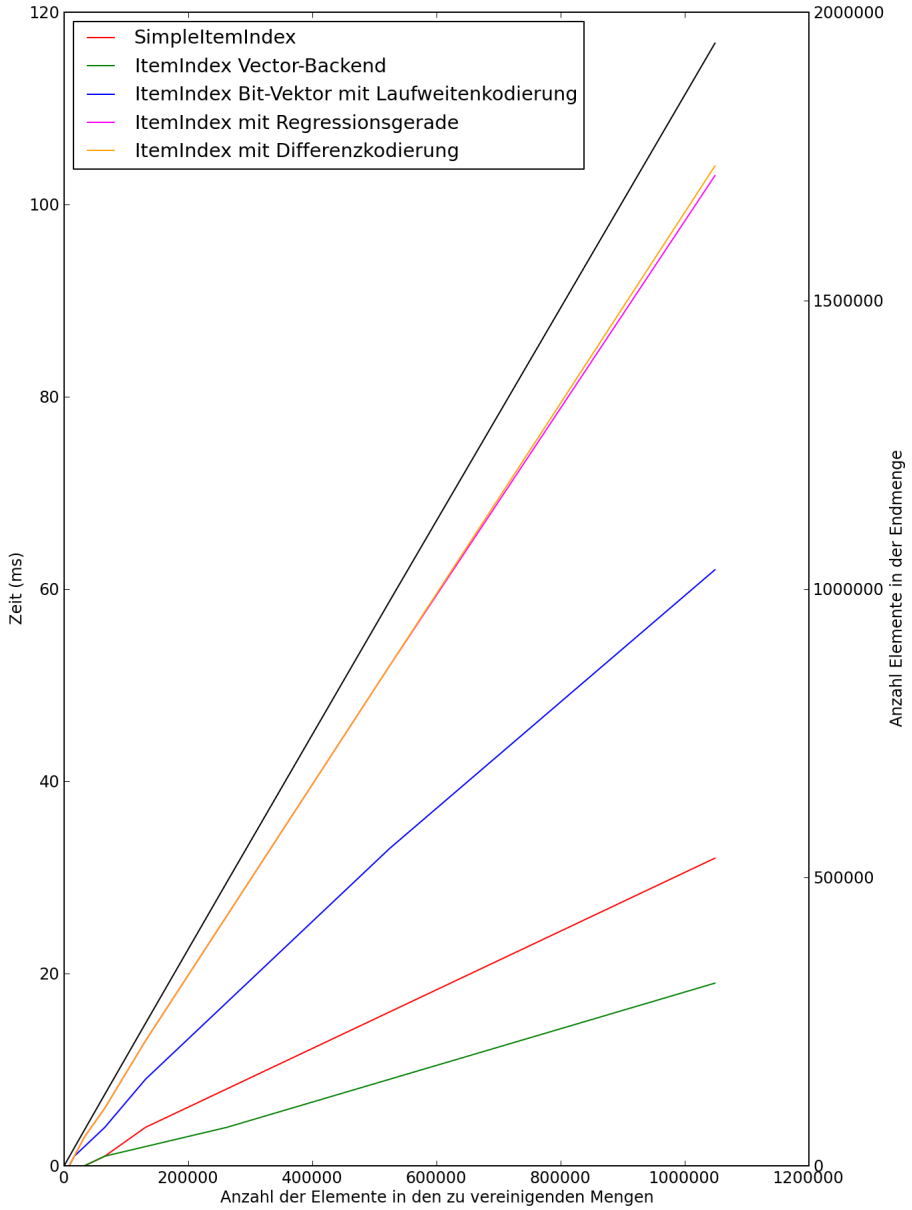


Abbildung 6.2: Vereinigung zweier Mengen für unterschiedliche ItemIndex Speicherschemata auf dem Desktoprechner, Größe der Ergebnismenge in Schwarz

Listing 6.1 Simulation einer Komplettierungsanfrage ohne Mengenoperationen

Eingabe Eingabe der Zeichenketten eines Elements der Datenbank, **k** minimale Zeichenkettenlänge, **n** Anzahl der Teilzeichenketten, die aus diesem Element erzeugt werden sollen, die zur Erzeugung der Komplettierungsanfragen genutzt werden

Teilen Die Elementzeichenketten werden an zusätzlichen Trennzeichen aufgetrennt und die so entstehenden Teilzeichenketten werden zu den Elementzeichenketten hinzugefügt.

Beschneiden Um Teilzeichenketten für Zeichenketten ohne Trennzeichen zu erhalten, werden aus den vorhandenen Zeichenketten, neue Teilzeichenketten erzeugt

- Neue Länge = Minimale Länge + Zufallszahl
- Neuer Beginn = Zufallszahl zwischen 0 und (Länge - NeueLänge)
- Füge neue Teilzeichenkette hinzu

Filtern Entferne alle Zeichenketten, die kleiner sind als **k**

Auswählen Wähle aus den verbliebenen Zeichenketten zufällig **n** Zeichenketten für die Komplettierung

Simulation Für jede so erhaltene Zeichenkette, führe folgende Simulation aus

1. Erstelle ein ItemSet mit den ersten k Buchstaben der ersten Zeichenkette
2. Aktualisiere das ItemSet um den jeweils nächsten Buchstaben der Zeichenkette, bis es keine weiteren neuen Buchstaben gibt

verwendet einen PAQ-Kompressionsalgorithmus und benötigt relativ lange zur Kompression und Dekompression. Lzop hingegen ist ein Echtzeitkompressionsalgorithmus, welcher vor allem auf Geschwindigkeit optimiert ist. Hier zeigt sich für alle Datensätze ein hohes Maß an Kompressibilität bei Verwendung des Bit-Vektor- oder differenzkodierten ItemIndex. Um die zu speichernden Daten weiter zu verkleinern könnte daher z. B. eine transparente Kompression auf Basis des lzo-Algorithmus getestet werden.

	BW	DE	GB	IT	ES	NL
Daten einlesen	17 s	121 s	43 s	42 s	30 s	39 s
Grenzpolygone schneiden	53 s	169 s	65 s	146 s	50 s	11 s
Trie erstellen	19 S	134 s	63 s	34 s	63 s	15 s
Trie mit vollem Index erstellen	36 s	352 s	107 s	59 s	95 s	19 s
Maximaler Speicherverbrauch	2,3 GiB	11,2 GiB	6,4 GiB	3,4 GiB	5,1 GiB	2,1 GiB

Tabelle 6.4: Verarbeitungszeiten verschiedener Datensätze. Für das Polygonschneiden wurde ein 500x500-Raster genutzt. Als Komplettierer wurde ein GST mit vollem Index erstellt.

Listing 6.2 Simulation einer Komplettierungsanfrage mit Mengenoperationen (Schnitt)

Eingabe Eingabe der Zeichenketten eines Elements der Datenbank, **k** minimale Zeichenkettenlänge, **n** Anzahl der Teilzeichenketten, die aus diesem Element erzeugt werden sollen, die zur Erzeugung der Komplettierungsanfragen genutzt werden

Teilen Die Elementzeichenketten werden an zusätzlichen Trennzeichen aufgetrennt und die so entstehenden Teilzeichenketten werden zu den Elementzeichenketten hinzugefügt.

Beschneiden Um Teilzeichenketten für Zeichenketten ohne Trennzeichen zu erhalten, werden aus den vorhanden Zeichenketten neue Teilzeichenketten erzeugt

- Neue Länge = Minimale Länge + Zufallszahl
- Neuer Beginn = Zufallszahl zwischen 0 und (Länge - NeueLänge)
- Füge neue Teilzeichenkette hinzu

Filtern Entferne alle Zeichenketten, die kleiner sind als **k**

Auswählen Wähle aus den verbliebenen Zeichenketten zufällig **n** Zeichenketten für die Komplettierung

Simulation Führe Simulation mit den so erhaltenen Zeichenketten aus

1. Erstelle ein ItemSet mit den ersten k Buchstaben der ersten Zeichenkette
2. Aktualisiere das ItemSet um den jeweils nächsten Buchstaben der Zeichenkette
3. Gibt es keine weiteren Buchstaben, aktualisiere mit den ersten k Buchstaben der nächsten Zeichenkette und gehe zu 2

	Datenbank	Zeichenkettenindex	TagStore	Trie	IndexStore
Deutschland	286 MiB	22 MiB	88 KiB	249 MiB	3,4 GiB
Großbritannien	147 MiB	13 MiB	62 KiB	98 MiB	1,3 GiB
Italien	98 MiB	6,9 MiB	29 KiB	73 MiB	509 MiB
Spanien	73 MiB	11 MiB	23 KiB	118 MiB	714 GiB
Niederlande	71 MiB	3,2 MiB	24 KiB	26 MiB	429 MiB
Baden-Württemberg	41MiB	3,5 MiB	32 KiB	40 MiB	363 MiB

Tabelle 6.5: Dateigrößen für einige Datensätze mit vollem Regressionsgeraden-ItemIndex. Dazu wurden noch die gleichen Zeichenketten ohne diakritische Zeichen hinzugefügt.

	# Zeichenketten	∅ Zeichenkettenlänge	⊤ Zeichenkettenlänge	# Elemente	$\lvert Alphabet \rvert$	# Tags
Deutschland	1.086.337	17,19	291	3.555.164	258	3315
Großbritannien	675.780	16,22	211	1.843.909	142	2373
Italien	351.736	17,18	165	908.238	136	1141
Spanien	586.668	16,03	140	1.150.610	137	896
Niederlande	188.816	14	104	1.055.659	121	939
Baden-Württemberg	183.594	16,59	142	514.220	125	1247

Tabelle 6.6: Statistiken zum Inhalt einiger Datensätze

	Regressionsgerade			Bit-Vektor			Differenzkodierung		
Kompression	ohne	lrzip	lzop	ohne	lrzip	lzop	ohne	lrzip	lzop
Deutschland	3,4 GiB	2,1 GiB	3,0 GiB	2,6 GiB	444 MiB	1,1 GiB	2,6 GiB	386 MiB	840 MiB
Großbritannien	1,3 GiB	764 MiB	1,1 GiB	938 MiB	153 MiB	412 MiB	938 MiB	153 MiB	412 MiB
Italien	508 MiB	356 MiB	438 MiB	422 MiB	81 MiB	175 MiB	422 MiB	81 MiB	175 MiB
Spanien	714 MiB	530 MiB	655 MiB	685 MiB	122 MiB	276 MiB	685 MiB	122 MiB	276 MiB
Niederlande	429 MiB	276 MiB	370 MiB	239 MiB	43 MiB	104 MiB	239 MiB	43 MiB	104 MiB
Baden-Württemberg	363 MiB	188 MiB	290 MiB	265 MiB	41 MiB	116 MiB	265 MiB	43 MiB	116 MiB

Tabelle 6.7: Statistiken zur Kompressibilität und Entropie einiger Datensätze mit vollem ItemIndex. Dazu wurden noch die gleichen Zeichenketten ohne diakritische Zeichen hinzugefügt.

6.3 Ausgewählte Datensätze

In diesem Abschnitt werden der Deutschland- sowie der Baden-Württemberg-Datensatz näher untersucht. Dabei wurde versucht möglichst viele verschiedene Optionen miteinander zu vergleichen. Es hat sich hierbei gezeigt, dass die Datenstruktur zur Suche des passenden ItemIndex einen unerheblichen Anteil an der Gesamtzeit der Suche besitzt. Der Unterschied zwischen GST und FlatGST spielte gegenüber den Kosten für die Mengenoperationen eine untergeordnete Rolle. Zu erwarten wäre, dass der GST mit kleinem Alphabet und großer Knotenanzahl schnellere Ergebnisse liefert als der FlatGST. Umgekehrt sollte der FlatGST für ein großes Alphabet und kleiner Anzahl an Knoten schneller sein. Der GST benötigte etwas mehr Platz, bietet jedoch die zusätzliche Information der nachfolgenden Zeichen für eine gegebene Komplettierungszeichenkette. Die Geschwindigkeit der Mengenoperationen hing vor allem vom verwendeten ItemIndex-Typ ab.

6.3.1 Deutschland

Deutschland ist der größte der untersuchten Datensätze. Je nach Komplettierer und ItemIndex-Speicherschemata variiert die Dateigröße für den ItemIndexStore zwischen 343 MiB und 3,4 GiB. Die Dateigröße der Komplettierer zeigt keine große Variation, da sich hier lediglich die ItemIndex-Referenzen ändern können.

Tabelle 6.9 zeigt die Komplettierungszeiten für die drei unterschiedlichen Komplettierer mit Regressionsgeraden-Index. Dabei zeigt sich, dass die Unterschiede der mittleren Komplettierungszeiten von FlatGST und GST mit gleichem ItemIndex-Typ gegenüber den Kosten für die Mengenoperationen nicht so stark ins Gewicht fallen. Wie zu erwarten, benötigt der FlatGST mit Zeichenkettenreferenz am längsten, da dieser die gesamte Datenbank nach Elementen mit passenden Zeichenkettenreferenz durchsuchen muss. Demgegenüber steht jedoch eine um Faktor 10 kleinere ItemIndexStore-Datei. Die hohen Komplettierungszeiten für den indirekten ItemIndex dürften mehrere Ursachen haben. So steigt die Anzahl an Indirektionen proportional zur Länge der Komplettierungszeichenketten, was den Zugriff auf ein Element verlangsamt. Hinzu kommt die hierdurch verursachte Belastung sämtlicher Caches. Sowohl der CPU-Cache als auch der Arbeitsspeicher-Cache, der durch das ChunkedMmappedFile implementiert ist, dürfte hierdurch oft Cache-Seiten ersetzen müssen. So hat das ChunkedMmappedFile standardmäßig 16 Cache-Seiten, was bei einer 17-fachen Indirektion bei jedem Zugriff auf eine Indirektionsebene zu einer Ersetzung führt.

Tabelle 6.18 vergleicht die Komplettierungszeiten von FlatGST mit Zeichenkettenreferenzen und dem GST mit dynamischem ItemIndex. Hierfür wurden mehrere GST-Varianten erzeugt, welche durch die Optionen in der Index-Optionen-Zeile näher spezifiziert sind. Hier stehen die beiden Zahlen nach *nfr* für die Ebenen im Baum, die keinen Präfix- und Teilzeichenkettenindex erhalten, *nfpi* und *nfsi* steht für die maximale Anzahl von Vergleichen, um den Präfix- respektive Teilzeichenkettenindex für einen gegebenen Knoten zu erzeugen. Für den FlatGST mit Zeichenkettenreferenzen wurde zudem eine Variante mit Elementreferenzen getestet. Hier bedeutet *msl*=n, dass der Pfad im Trie mindestens Länge n besitzen muss,

	Größe	Header	Zeichenketten	Index-Zeiger
Trie	217.291.278	22.492.990 (10.4%)	112.140.614 (51,6 %)	82.657.674 (38,0 %)
FlatGST	182.533.281	20.808.794 (11,4 %)	51.839.451 (28,4 %)	109.885.036 (60,2 %)
FlatGST mit Zeichenkettenreferenzen	251.833.255	20.902.160 (8.3 %)	51.877.650 (20,6 %)	179.053.445 (71,1 %)

Tabelle 6.8: Deutschland-Datensatz: Größenverteilung der Information für verschiedene Komplettierer.

damit ein Element-ItemIndex erstellt wird. Mit der Option nid=m lässt sich noch einstellen, dass Knoten, deren Element-ItemIndex kleiner als m ist, einen Element-ItemIndex erhalten sollen. In diesem Szenario ist der FlatGST dem GST in nahezu jeder Variante überlegen. Auch mit anderen ItemIndex-Typen wie dem Bit-Vektor-basierten ItemIndex, der in Tabelle 6.11 aufgeführt ist, stellte sich der FlatGST mit Zeichenkettenreferenzen als die schnellere Variante heraus.

Tabelle 6.11 vergleicht die Komplettierungszeiten und ItemIndexStore Größen verschiedener ItemIndex-Typen. Der Bit-Vektor-basierte ItemIndex sticht vor allem in Verbindung mit dynamischem ItemIndex heraus. Nutzt man hingegen einen vollen oder merge ItemIndex, so ist der differenzkodierte ItemIndex schneller, was vor allem der Verringerung der Größe der Ergebnismengen bei jedem Mengenschnitt geschuldet sein dürfte. Der Regressionsgeraden-basierte ItemIndex kann jedoch, trotz der möglichen Nutzung einer Binärsuche bei Schnittoperationen, mit den anderen beiden Varianten nicht konkurrieren.

Tabelle 6.18 zeigt die Komplettierungszeiten bei Verwendung der Iterator-Schnittstelle mit einem Regressionsgeraden-basierten ItemIndex. Die Komplettierungszeiten dieser Variante sind für den Deutschland-Datensatz so hoch, dass sich deren Nutzung auf dem Mobiltelefon verbietet.

Abschließend zeigt Tabelle 6.19 die Auswirkungen verschiedener Suffix-Trennzeichen auf die Datei-Größe sowie die Komplettierungszeiten mit Regressionsgeraden-basiertem ItemIndex. Nutzt man einen eingeschränkten Satz an Suffix-Trennzeichen (-sd), so verringert sich die Dateigröße des ItemIndexStore massiv. Auch die Komplettierungszeiten verringern sich, da die Größe der Ergebnismengen abnimmt.

	Trie			FlatGST		FlatGST mit Zeichenkettenreferenzen	
Indextyp	merge	voller	indirekter	merge	voller	merge	voller
Erstellzeit (Min:Sek)	14:36	15:08	52:52	15:01	15:57	11:15	11:05
Größe Komplettierer	207 MiB	208 MiB	185 MiB	145 MiB	175 MiB	212 MiB	241 MiB
Größe ItemIndexStore	2,9 GiB	3,3 GiB	2,5 GiB	2,9 GiB	3,3 GiB	278 MiB	317 MiB
Anzahl Indices	3.640.128	3.984.830	2.208.889	3.640.128	3.984.830	3.648.725	3.999.138
Gesamtentropie	21,69	21,70	19,81	21,69	21,70	20,23	20,23
Diskrete Gesamtentropie	22,16	22,16	20,29	22,16	22,16	20,75	20,73
Mittlere # benötigter Bits	19,42	19,52	13,25	19,42	19,52	19,96	20,02
Komplettierungszeiten mit Mengenoperationen auf dem Mobiltelefon							
Minimum	218 ms	81 ms	1639 ms	568 ms	317 ms	14815 ms	13153 ms
Maximum	7733 ms	73041 ms	1183483 ms	18684 ms	75780 ms	43995 ms	30597 ms
Mittelwert	4089 ms	8820 ms	176729 ms	8663 ms	9322 ms	23747 ms	15667 ms
Standardabweichung	2357 ms	21560 ms	355503 ms	6829 ms	22302 ms	10505 ms	5047 ms
Median	4035 ms	784 ms	21628 ms	7004 ms	760 ms	19127 ms	13687 ms
Komplettierungszeiten ohne Mengenoperationen auf dem Mobiltelefon							
Minimum	41 ms	0,5 m	0,9 ms	136 ms	0,5 ms	12693 ms	12486 ms
Maximum	11542 ms	101 ms	197 ms	16401 ms	2821 ms	40780 ms	43524 ms
Mittelwert	1521 ms	54 ms	71 ms	1836 ms	125 ms	16130 ms	16027 ms
Standardabweichung	2113 ms	30 ms	31 ms	2436 ms	285 ms	6209 ms	6240 ms
Median	490 ms	60 ms	65 ms	792 ms	85 ms	14340 ms	14306 ms
Komplettierungszeiten mit Mengenoperationen auf dem Desktoprechner							
Minimum	53 ms	51 ms	711 ms	88 ms	82 ms	512 ms	531 ms
Maximum	323 ms	722 ms	60382 ms	395 ms	733 ms	1138 ms	1529 ms
Mittelwert	162 ms	216 ms	18206 ms	219 ms	263 ms	632 ms	680 ms
Standardabweichung	86 ms	183 ms	20392 ms	103 ms	172 ms	170 ms	284 ms
Komplettierungszeiten ohne Mengenoperationen auf dem Desktoprechner							
Minimum	0,6 ms	0,01 ms	0,4 ms	0,8 ms	0,03 ms	421 ms	418 ms
Maximum	684 ms	33 ms	48 ms	1112 ms	148 ms	598 ms	613 ms
Mittelwert	74 ms	13 ms	15 ms	119 ms	24 ms	503 ms	504 ms
Standardabweichung	98 ms	9 ms	11 ms	200 ms	23 ms	32 ms	34 ms

Tabelle 6.9: Deutschland-Datensatz: Übersicht über Komplettierungszeiten für verschiedene Komplettierer

	Trie					FlatGST mit Zeichenkettenreferenzen	
Index-Optionen	voller	nfr=(0, 3)	nfr=(4, 255)	nfr=(0, 255)	merge, nfpi=10000, nfsi=10000		msl=3, nid=10^{10}
Größe Komplettierer	185 MiB	207 MiB	154 MiB	154 MiB	154 MiB	241 MiB	175 MiB
Größe ItemIndexStore	3,3 GiB	2,1 GiB	1,6 GiB	373 MiB	2.7 GiB	317 MiB	2,2 GiB
Anzahl Indices	3.640.128	3.942.468	1.643.269	1.597.021	1.631.537	3.999.138	3.641.367
Gesamtentropie	21,69	21,69	21,73	21.73	21,70	20,23	21.66
Diskrete Gesamtentropie	22,16	22,16	22,14	22,16	22,16	20,73	22.17
Mittlere # benötigter Bits	19,42	19,72	19,32	19,75	19,30	20,02	19.39
Komplettierungszeiten mit Mengenoperationen auf dem Mobiltelefon							
Minimum	81 ms	1287 ms	1286 ms	2769 ms	761 ms	13153 ms	379 ms
Maximum	73041 ms	30159 ms	173752 ms	177493 ms	7024 ms	30597 ms	15318 ms
Mittelwert	8820 ms	17958 ms	80084 ms	83990 ms	4104 ms	15667 ms	11078 ms
Standardabweichung	21560 ms	9813 ms	64465 ms	65096 ms	1965 ms	5047 ms	5439 ms
Komplettierungszeiten ohne Mengenoperationen auf dem Mobiltelefon							
Minimum	0,5 ms	157 ms	29 ms	293 ms	41 ms	12486 ms	133 ms
Maximum	1278 ms	30305 ms	177033 ms	201760 ms	7716 ms	43524 ms	44176 ms
Mittelwert	64 ms	6217 ms	25314 ms	29471 ms	1510 ms	16027 ms	4810 ms
Standardabweichung	132 ms	8210 ms	48705 ms	50170 ms	1863 ms	6240 ms	8150 ms
Median	35 ms	2079 ms	37840 ms	7104 ms	630 ms	14306 ms	841 ms
Komplettierungszeiten mit Mengenoperationen auf dem Desktoprechner							
Minimum	51 ms	251 ms	507 ms	548 ms	100 ms	531 ms	75 ms
Maximum	722 ms	4114 ms	4366 ms	4095 ms	344 ms	1527 ms	4565 ms
Mittelwert	216 ms	868 ms	2093 ms	2002 ms	222 ms	679 ms	929 ms
Standardabweichung	183 ms	1117 ms	1349 ms	1251 ms	85 ms	284 ms	1250 ms
Komplettierungszeiten ohne Mengenoperationen auf dem Desktoprechner							
Minimum	0,01 ms	0,5 ms	0,06 ms	0,08 ms	0,6 ms	418 ms	0,03 ms
Maximum	33 ms	680 ms	3952 ms	4357 ms	310 ms	613 ms	562 ms
Mittelwert	13 ms	193 ms	980 ms	711 ms	76 ms	503 ms	149 ms
Standardabweichung	9 ms	144 ms	158 ms	1047 ms	59 ms	34 ms	200 ms

Tabelle 6.10: Deutschland-Datensatz: Übersicht über die Komplettierungszeiten für verschiedene ItemIndex-Speicherschemata.

Optionen		voller	merge	nfr=(0,3)	nfr=(4,255)	nfr=(0,255)	merge, nfpi=10000, nfsi=10000
ItemIndexStore	Regressionsgerade	3,3 GiB	2,9 GiB	2,1 GiB	1,6 GiB	373 MiB	2,7 GiB
	Bit-Vektor	2,4 GiB	2,1 GiB	1,9 GiB	864 MiB	343 MiB	1,9 GiB
	Differenzkodierung	2,4 GiB	2,1 GiB	1,9 GiB	864 MiB	343 MiB	1,9 GiB
Minimum	Regressionsgerade	81 ms	218 ms	1.287 ms	1.286 ms	2.769 ms	761 ms
	Bit-Vektor	110 ms	247 ms	1.209 ms	1.844 ms	3.756 ms	485 ms
	Differenzkodierung	88 ms	562 ms	2.219 ms	2.879 ms	5.575 ms	708 ms
Maximum	Regressionsgerade	73.041 ms	7.733 ms	30.159 ms	173.752 ms	177.493 ms	7.024 ms
	Bit-Vektor	10.598 ms	3.310 ms	14.790 ms	46.516 ms	58.352 ms	1.741 ms
	Differenzkodierung	1.732 ms	2.643 ms	27.369 ms	361.654 ms	455.205 ms	2.572 ms
Mittelwert	Regressionsgerade	8.820 ms	4.089 ms	17.958 ms	80.084 ms	83.990 ms	4.104 ms
	Bit-Vektor	2.319 ms	1.462 ms	4.276 ms	1.631 ms	37.137 ms	1.024 ms
	Differenzkodierung	368 ms	1.602 ms	6.772 ms	162.587 ms	220.054 ms	1.659 ms
Standardabweichung	Regressionsgerade	21.560 ms	2.357 ms	9.813 ms	64.465 ms	65.096 ms	1.965 ms
	Bit-Vektor	3.476 ms	920 ms	3.986 ms	571 ms	22.193 ms	366 ms
	Differenzkodierung	493 ms	756 ms	7.062 ms	135.798 ms	178.339 ms	645 ms

Tabelle 6.11: Deutschland-Datensatz: Vergleich der Komplettierungszeiten mit Mengenoperationen von Bit-Vektor-basiertem und Regressionsgeraden-basiertem ItemIndex auf dem Mobiltelefon

	Trie	FlatGST	FlatGST mit Zeichenkettenreferenzen	
Index-Optionen	voller	merge	voller	msl=3, nid=10^{10}
Größe des ItemIndexStore	3,3 GiB	2,9 GiB	317 MiB	2,2 GiB
Komplettierungszeiten auf dem Mobiltelefon				
Minimum	1213 ms	1287 ms	1286 ms	1.081 ms
Maximum	73041 ms	30159 ms	173752 ms	69.140 ms
Mittelwert	8820 ms	17958 ms	80084 ms	26.173 ms
Standardabweichung	21560 ms	9813 ms	64465 ms	24.009 ms
Komplettierungszeiten auf dem Desktoprechner				
Minimum	51 ms	251 ms	507 ms	548 ms
Maximum	722 ms	4114 ms	4366 ms	4095 ms
Mittelwert	216 ms	868 ms	2093 ms	2002 ms
Standardabweichung	183 ms	1117 ms	1349 ms	1251 ms

Tabelle 6.12: Deutschland-Datensatz: Komplettierungszeiten mit Mengenoperationen mit der Iterator-Schnittstelle

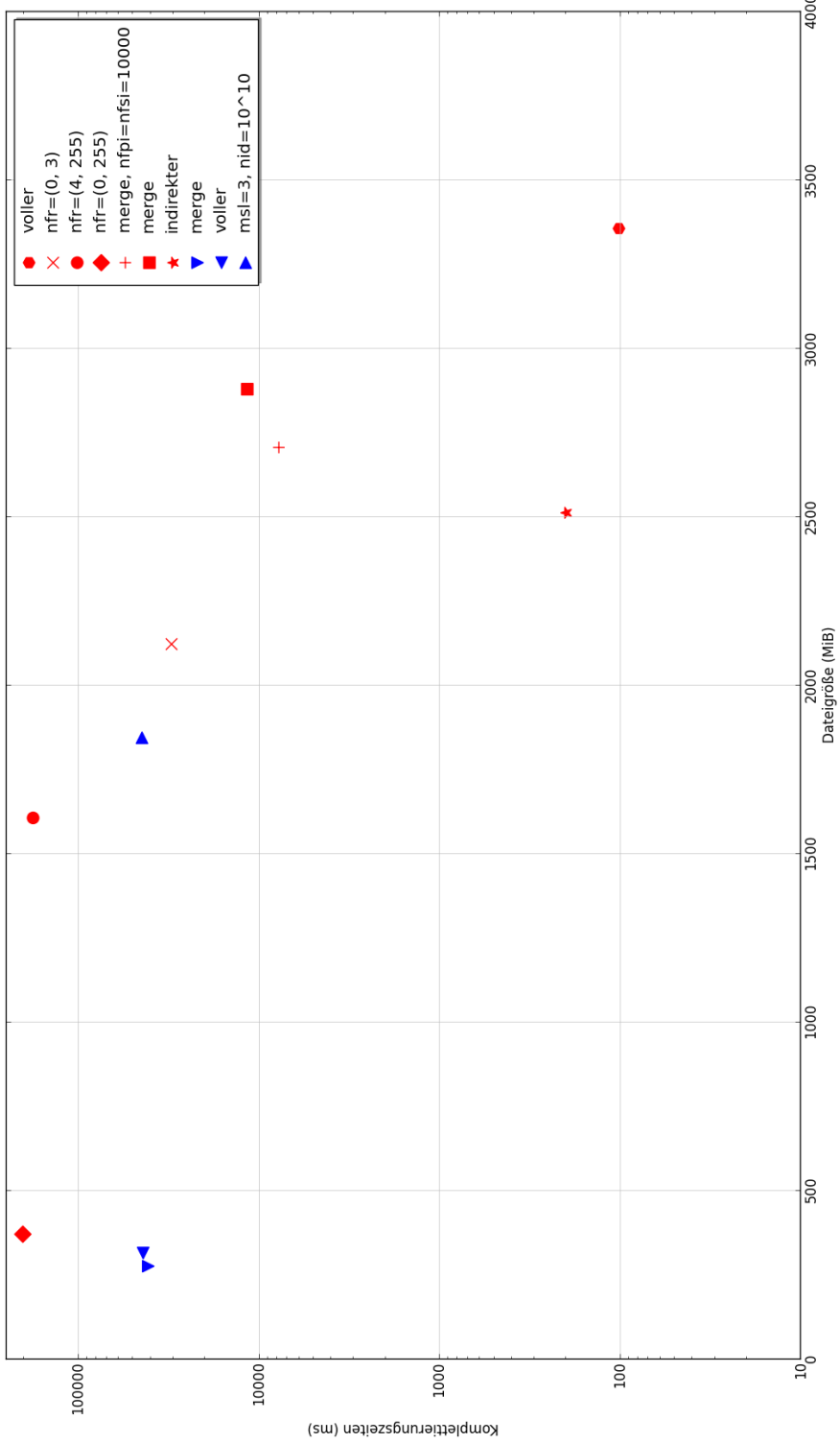


Abbildung 6.3: Deutschland-Datensatz: Maximale Komplettierungszeiten ohne Mengenoperationen in Abhängigkeit von der Dateigröße des ItemIndexStores mit verschiedenen Komplettierern mit Regressionsgeraden-Index auf dem Mobiltelefon

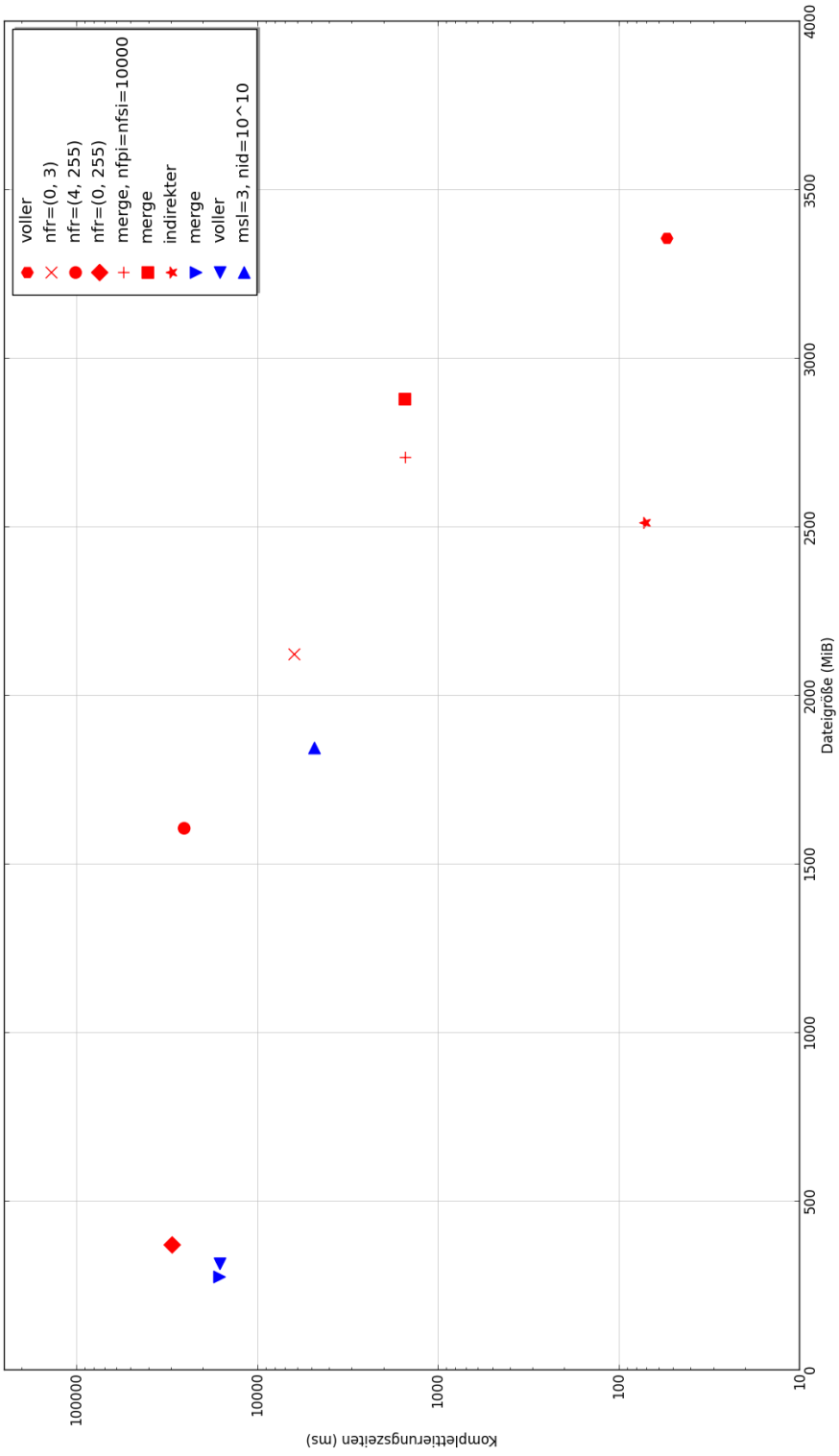


Abbildung 6.4: Deutschland-Datensatz: Mittlere Komplettierungszeiten ohne Mengenoperationen in Abhängigkeit von der Dateigröße des ItemIndexStores mit verschiedenen Komplettierern mit Regressionsgeraden-Index auf dem Mobiltelefon

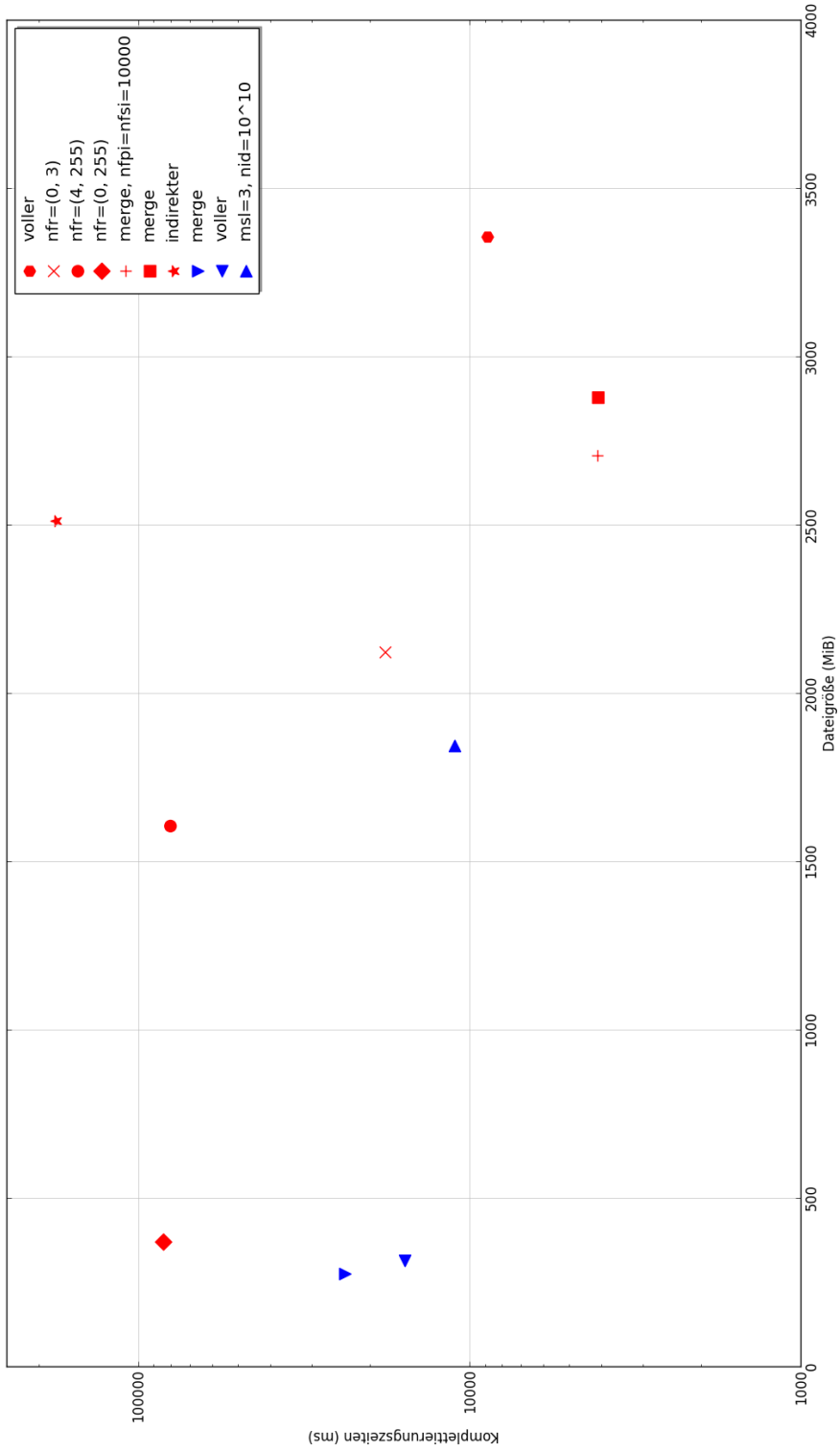


Abbildung 6.5: Deutschland-Datensatz: Mittlere Komplettierungszeiten mit Mengenoperationen in Abhängigkeit von der Dateigröße des ItemIndexStores mit verschiedenen Komplettierern mit Regressionsgeraden-Index auf dem Mobiltelefon

	-s -adi	-s	-sd
Daten einlesen	121 s	121 s	121 s
Grenzpolygone schneiden	169 s	169 s	169 s
Trie erstellen	134 s	98 s	23 s
Trie mit vollem Index erstellen	352 s	290 s	53 s
Maximaler Speicherverbrauch	11,2 GiB	10,1 GiB	4,5 GiB
Knotenanzahl	13.615.470	11.227.449	1.684.430
Größe Trie	249MiB	208 MiB	33 MiB
Größe ItemIndexStore	3,4 GiB	3,2 GiB	706 MiB
Komplettierungszeiten mit Mengenoperationen auf dem Mobiltelefon			
Minimum	80 ms	81 ms	61 ms
Maximum	72950 ms	73041 ms	478 ms
Mittelwert	8883 ms	8820 ms	195 ms
Standardabweichung	21508 ms	21560 ms	142 ms
Median	773 ms	784 ms	162 ms
Komplettierungszeiten ohne Mengenoperationen auf dem Mobiltelefon			
Minimum	0,4 ms	0,5 ms	0,3 ms
Maximum	1278 ms	101 ms	483 ms
Mittelwert	64 ms	54 ms	35 ms
Standardabweichung	132 ms	30 ms	53 ms
Median	35 ms	60 ms	32 ms
Komplettierungszeiten mit Mengenoperationen auf dem Desktoprechner			
Minimum	42 ms	51 ms	27 ms
Maximum	4572 ms	722 ms	178 ms
Mittelwert	859 ms	216 ms	73 ms
Standardabweichung	1356 ms	183 ms	55 ms
Median	308 ms	177 ms	43 ms
Komplettierungszeiten ohne Mengenoperationen auf dem Desktoprechner			
Minimum	0,02 ms	0,01 ms	0,01 ms
Maximum	62 ms	33 ms	27 ms
Mittelwert	12 ms	13 ms	9 ms
Standardabweichung	10 ms	9 ms	7 ms

Tabelle 6.13: Deutschland-Datensatz: Erstell- und Komplettierungszeiten mit verschiedenen Suffix-Optionen mit Regressionsgeraden-basiertem ItemIndex

Kompression

6.3.2 Baden-Württemberg

Der Baden-Württemberg-Ausschnitt ist im Vergleich zum Deutschland-Ausschnitt deutlich kleiner. Die Größe des Komplettierers bewegt sich dabei im Bereich zwischen 40 MiB und 5,3 MiB. Die ItemIndexStore-Größe variiert zwischen 36 MiB und 363 MiB. Tabelle 6.14 gibt einen Überblick über die Verteilung der Informationen drei verschiedener Komplettierer. Vergleicht man den Trie mit dem FlatGST, fällt sofort auf, dass der Trie deutlich mehr Speicherplatz für das Speichern der Zeichenketten benötigt. Darin eingeschlossen sind die Kindknotenzeiger, welche jedoch lediglich 14,5 Prozent der Dateigröße ausmachen. Der FlatGST benötigt demgegenüber mehr Platz für die ItemIndex-Zeiger. Der geringe Unterschied zwischen FlatGST und Trie zeigt darüberhinaus, dass die Verwendung einer expliziten Baumstruktur für diesen Datensatz keine signifikanten Nachteile besitzt.

Daten über den ItemIndexStore sind in Tabelle 6.15 und 6.16 aufgeführt. Dabei zeigte sich, dass die Verwendung einer Regressionsgeraden zur Speicherung der Mengen gerechtfertigt ist, da für jeden Komplettierer die benötigte Speichermenge unter der Entropie der Eingabedaten liegt. Hervor stach hierbei der indirekte ItemIndex, da dieser im Vergleich zu den anderen Speicherschemata deutlich unter der Entropie der Eingabedaten liegt. Die Idee der besseren Approximierung durch eine homogenere Verteilung der Daten scheint zumindest für diesen Datensatz gute Ergebnisse zu liefern. Darüberhinaus ist der Anteil der gleichen ItemIndexe im ItemIndexStore gegenüber den anderen Speicherschemata deutlich geringer.

Die Komplettierungszeiten sind in Tabelle 6.15 und 6.16 jeweils für Komplettierungen mit Mengenoperationen und Komplettierungen ohne Mengenoperationen dargestellt. Vergleicht man den Trie mit dem FlatGST fällt auf, dass der Trie etwas schneller ist als der FlatGST. Dies dürfte darin begründet sein, dass die Alphabetgröße lediglich 122 Zeichen betrug, die Anzahl der Elemente jedoch nahezu 200.000. Der FlatGST benötigt hierbei in jedem Fall $\log_2(188000) \approx 18$ Schritte bis zum Endknoten, der Trie $\log_2(122) \approx 6 \cdot |Zeichenkette|$. Allerdings springt der FlatGST am Anfang der Suche stärker in der Datei hin und her, was zu einer höheren Anzahl an Cache-Misses führen dürfte. Im Trie hingegen dürfte die Suche in 6 Schritten innerhalb einer Cache-Zeile stattfinden. Darüberhinaus nimmt die Anzahl an Kindknoten mit der Tiefe schnell ab. Die Unterschiede sind für den Benutzer jedoch unerheblich, da diese keine spürbare Größe erreichen. Desweiteren erkennt man, dass das Speicherschema der ItemIndexe den größten Einfluss auf die Komplettierungszeiten hat. Abbildung 6.6 zeigt den Einfluss der Dateigröße auf die maximale Komplettierungszeit, Abbildung 6.6 dagegen den Einfluss der Dateigröße auf die durchschnittliche Komplettierungszeit. Die Rangfolge der unterschiedlichen Speicherschemata bleibt hierbei größtenteils erhalten. Weit abgeschlagen bleiben in beiden Fällen die Speichervarianten ohne zusätzliche ItemIndices in höheren Ebenen des Baumes. Hier ist eine Suche mit dem FlatGST mit Zeichenkettenreferenz deutlich überlegen. Würde man die Iterator-Schnittstelle benutzen, würde sich der Vorteil des FlatGST nochmals verstärken. Vergleicht man die Komplettierungszeiten mit und ohne Mengenoperationen fällt vor allem die starke Erhöhung der Komplettierungszeit für

den indirekten Index auf. Für eine einfach Komplettierung ohne Zugriff auf die Elemente unterscheiden sich die Komplettierungszeiten nicht merklich vom vollen Index und liegen sogar unter dem vom Merge-ItemIndex. Werden jedoch Mengenoperationen durchgeführt, so dreht sich die Rangfolge um. Der indirekte ItemIndex ist nun deutlich langsamer als der Merge-ItemIndex. Dies dürfte vor allem daran liegen, dass die Anzahl an Indirektionen bis zur korrekten Elementreferenz mit der Länge der Suchzeichenkette steigt.

Tabelle 6.19 zeigt Dateigrößen und Komplettierungszeiten für unterschiedliche Suffixoptionen. Wie zu erwarten, war der Komplettierer mit speziellen Suffixtrennzeichen deutlich kleiner als der normale Suffix-Trie. Auch die Komplettierungszeiten waren etwas verringert.

Tabelle 6.17 zeigt die Komplettierungszeiten unterschiedlicher ItemIndex-Speicherschemata. Als Komplettierer wurde nur der Trie gewählt, da sowohl der Bit-Vektor-basierte als auch der differenzkodierte ItemIndex keinen wahlfreien Zugriff ermöglichen. Es zeigte sich, dass der Speicherverbrauch von Bit-Vektor und Differenzkodierung nahezu identisch und deutlich kleiner als die Regressionsgeraden-basierte Speicherung ausfällt. Dies macht sich auch bei der Geschwindigkeit der Mengenoperationen bemerkbar. Die Regressionsgeraden-basierte Speicherung dürfte hier ihren Vorteil nur dann ausspielen können, wenn sehr große mit sehr kleinen Mengen geschnitten werden sollen, denn dann kann eine Binärsuche genutzt werden. Am schnellsten ist hier der differenzkodierte ItemIndex, sofern die benötigten Daten nicht vom Unterbaum des Endknotens der Komplettierungszeichenkette eingesammelt werden müssen. In diesem Fall ist das Bit-Vektor-Speicherschema den anderen Speicherschemata deutlich überlegen.

	Größe	Header	Zeichenketten	ItemIndexStore-Zeiger
Trie	35.180.773	3.820.836 (10.9%)	19.096.373 (54,2 %)	12.263.564 (34,9 %)
FlatGST	28.270.925 Byte	3.392.511 (12 %)	8.113.755 (28,7 %)	16.764.658 (59,3 %)
FlatGST mit Zeichenkettenreferenzen	41.181.051 Byte	3.376.846 (8.2 %)	8.112.667 (19,7 %)	29.691.537 (72,1 %)

Tabelle 6.14: Größenverteilung der Information für verschiedene Komplettierer

	Trie			FlatGST		FlatGST mit Zeichenkettenreferenzen	
Indextyp	merge	voller	indirekter	merge	voller	merge	voller
Erstellzeit (Min:Sek)	2:15	2:14	6:10	2:16	2:19	1:57	1:57
Größe Komplettierer	34MiB	34 MiB	31 MiB	23 MiB	27 MiB	35 MiB	40 MiB
Größe ItemIndexStore	313 MiB	352 MiB	279 MiB	313MiB	352 MiB	36 MiB	41 MiB
Anzahl Indices	633.402	689.077	427.112	633.402	689.077	635.273	691.897
Gesamtentropie	18,86	18,87	17,65	18,86	18,87	17,67	17,67
Diskrete Gesamtentropie	19,35	19,37	18,1	19,35	19,37	18,18	18,17
Mittlere # benötigter Bits	17,07	16,99	11,67	17,07	16,99	17,73	17,77
Komplettierungszeiten mit Mengenoperationen auf dem Mobiltelefon							
Minimum	85 ms	60 ms	162 ms	107 ms	37 ms	1662 ms	1764 ms
Maximum	826 ms	728 ms	3422 ms	1145 ms	756 ms	2094 ms	2211 ms
Mittelwert	542 ms	270 ms	1297 ms	588 ms	275 ms	1875 ms	2021 ms
Standardabweichung	191 ms	233 ms	1196 ms	257 ms	241 ms	143 ms	161 ms
Komplettierungszeiten ohne Mengenoperationen auf dem Mobiltelefon							
Minimum	0,7 ms	0,3 ms	0,4 ms	4 ms	0,4 ms	1245 ms	1228 ms
Maximum	713 ms	264 ms	391 ms	775 ms	615 ms	2070 ms	2102 ms
Mittelwert	134 ms	36 ms	51 ms	147 ms	47 ms	1658 ms	1656 ms
Standardabweichung	155 ms	32 ms	51 ms	154 ms	70 ms	115 ms	112 ms
Median	64 ms	32 ms	46 ms	80 ms	33 ms	1639 ms	1649 ms

Tabelle 6.15: Baden-Württemberg-Datensatz: Übersicht über die Komplettierungszeiten für verschiedene Komplettierer

	Trie					FlatGST mit Zeichenkettenreferenzen	
Index-Optionen	voller	nfr=(0, 3)	nfr=(4, 255)	nfr=(0, 255)	merge, nfpi=10000, nfsi=10000		msl=3, nid=10^{10}
Größe Komplettierer	34 MiB	34 MiB	26 MiB	26 MiB	34 MiB	40 MiB	27 MiB
Größe ItemIndexStore	352 MiB	203 MiB	189 MiB	39 MiB	286MiB	36 MiB	242 MiB
Anzahl Indices	689.077	669.153	294.573	272.379	286.834	691.897	689.640
Gesamtentropie	18,87	18,85	18,90	18,91	18,85	17,67	18.86
Diskrete Gesamtentropie	19,37	19,34	19,35	19,37	19,35	18,17	19.35
Mittlere # benötigter Bits	16,99	17,25	16,81	17,54	16,84	17,77	17.08
Komplettierungszeiten mit Mengenoperationen auf dem Mobiltelefon							
Minimale Komplettierungszeit	60 ms	885 ms	292 ms	809 ms	393 ms	1764 ms	29 ms
Maximale Komplettierungszeit	728 ms	4449 ms	14873 ms	19927 ms	785 ms	2211 ms	782 ms
Mittlere Komplettierungszeit	270 ms	3288 ms	11372 ms	15065 ms	579 ms	2021 ms	313 ms
Standardabweichung	233 ms	1015 ms	4251 ms	5384 ms	119 ms	161 ms	245 ms
Komplettierungszeiten ohne Mengenoperationen auf dem Mobiltelefon							
Minimum	0,3 ms	3 ms	0,4 ms	1,6 ms	5 ms	1228 ms	0,4 ms
Maximum	264 ms	3280 ms	13661 ms	17816 ms	1024 ms	2102 ms	599 ms
Mittelwert	36 ms	580 ms	1515 ms	2496 ms	166 ms	1656 ms	48 ms
Standardabweichung	32 ms	773 ms	3111 ms	4518 ms	176 ms	112 ms	68 ms
Median	32 ms	218 ms	192 ms	456 ms	97 ms	1649 ms	33 ms

Tabelle 6.16: Baden-Württemberg-Datensatz: Übersicht über die Komplettierungszeiten für verschiedene ItemIndex-Speicherschemata

Optionen		voller	merge	nfr=(0,3)	nfr=(4,255)	nfr=(0,255)	merge, nfpi=10000, nfsi=10000
ItemIndexStore	Regressionsgerade	352 MiB	313 MiB	203 MiB	189 MiB	39 MiB	286MiB
	Bit-Vektor	252 MiB	225 MiB	188 MiB	101 MiB	36 MiB	184 MiB
	Differenzkodierung	252 MiB	225 MiB	188 MiB	101 MiB	38 MiB	184 MiB
Minimum	Regressionsgerade	60 ms	85 ms	885 ms	292 ms	809 ms	393 ms
	Bit-Vektor	40 ms	73 ms	196 ms	292 ms	713 ms	73 ms
	Differenzkodierung	38 ms	41 ms	292 ms	400 ms	1.317 ms	67 ms
Maximum	Regressionsgerade	728 ms	826 ms	4449 ms	14.873 ms	19.927 ms	785 ms
	Bit-Vektor	246 ms	275 ms	711 ms	2.493 ms	3.708 ms	378 ms
	Differenzkodierung	371 ms	193 ms	858 ms	14.946 ms	22.311 ms	380 ms
Mittelwert	Regressionsgerade	270 ms	542 ms	3288 ms	11.372 ms	15.065 ms	579 ms
	Bit-Vektor	88 ms	116 ms	366 ms	1.631 ms	2.338 ms	188 ms
	Differenzkodierung	95 ms	108 ms	532 ms	11.805 ms	17.440 ms	174 ms
Standardabweichung	Regressionsgerade	233 ms	191 ms	1.015 ms	4.251 ms	5.384 ms	119 ms
	Bit-Vektor	56 ms	56 ms	137 ms	571 ms	824 ms	102 ms
	Differenzkodierung	94 ms	39 ms	192 ms	3.932 ms	5.642 ms	90 ms

Tabelle 6.17: Vergleich der Komplettierungszeiten mit Mengenoperationen auf dem Mobiltelefon von Bit-Vektor-basiertem und Regressionsgeraden-basiertem ItemIndex für den Baden-Württemberg-Datensatz

	Trie	FlatGST	FlatGST mit Zeichenkettenreferenzen	
Index-Optionen	voller	merge	voller	msl=3, nid=10^{10}
Größe des ItemIndexStore	3,3 GiB	2,9 GiB	317 MiB	2,2 GiB
Komplettierungszeiten auf dem Mobiltelefon				
Minimum	33 ms	26 ms	15.525 ms	27 ms
Maximum	793 ms	1.506 ms	51.660 ms	814 ms
Mittelwert	311 ms	585 ms	29.152 ms	361 ms
Standardabweichung	271 ms	512 ms	11.996 ms	285 ms
Komplettierungszeiten auf dem Desktoprechner				
Minimum	51 ms	251 ms	507 ms	548 ms
Maximum	722 ms	4114 ms	4366 ms	4095 ms
Mittelwert	216 ms	868 ms	2093 ms	2002 ms
Standardabweichung	183 ms	1117 ms	1349 ms	1251 ms

Tabelle 6.18: Komplettierungszeiten mit Mengenoperationen mit dem Iterator-Interface mit dem Regressionsgeraden-basierten ItemIndex für den Baden-Württemberg-Datensatz

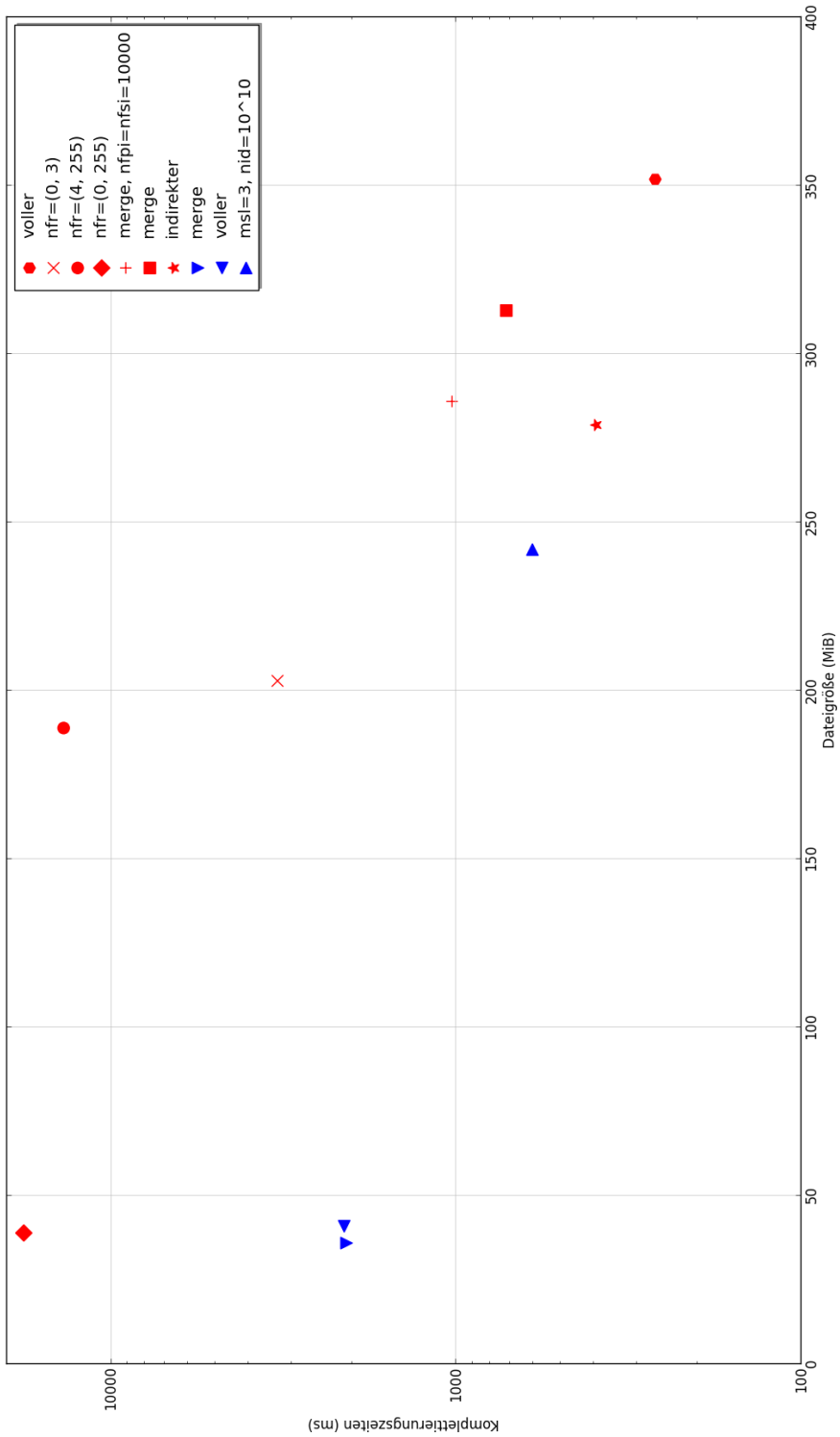


Abbildung 6.6: Baden-Württemberg-Datensatz: Maximale Komplettierungszeiten ohne Mengenoperationen in Abhängigkeit von der Dateigröße des ItemIndexStores mit verschiedenen Komplettierern auf dem Mobiltelefon

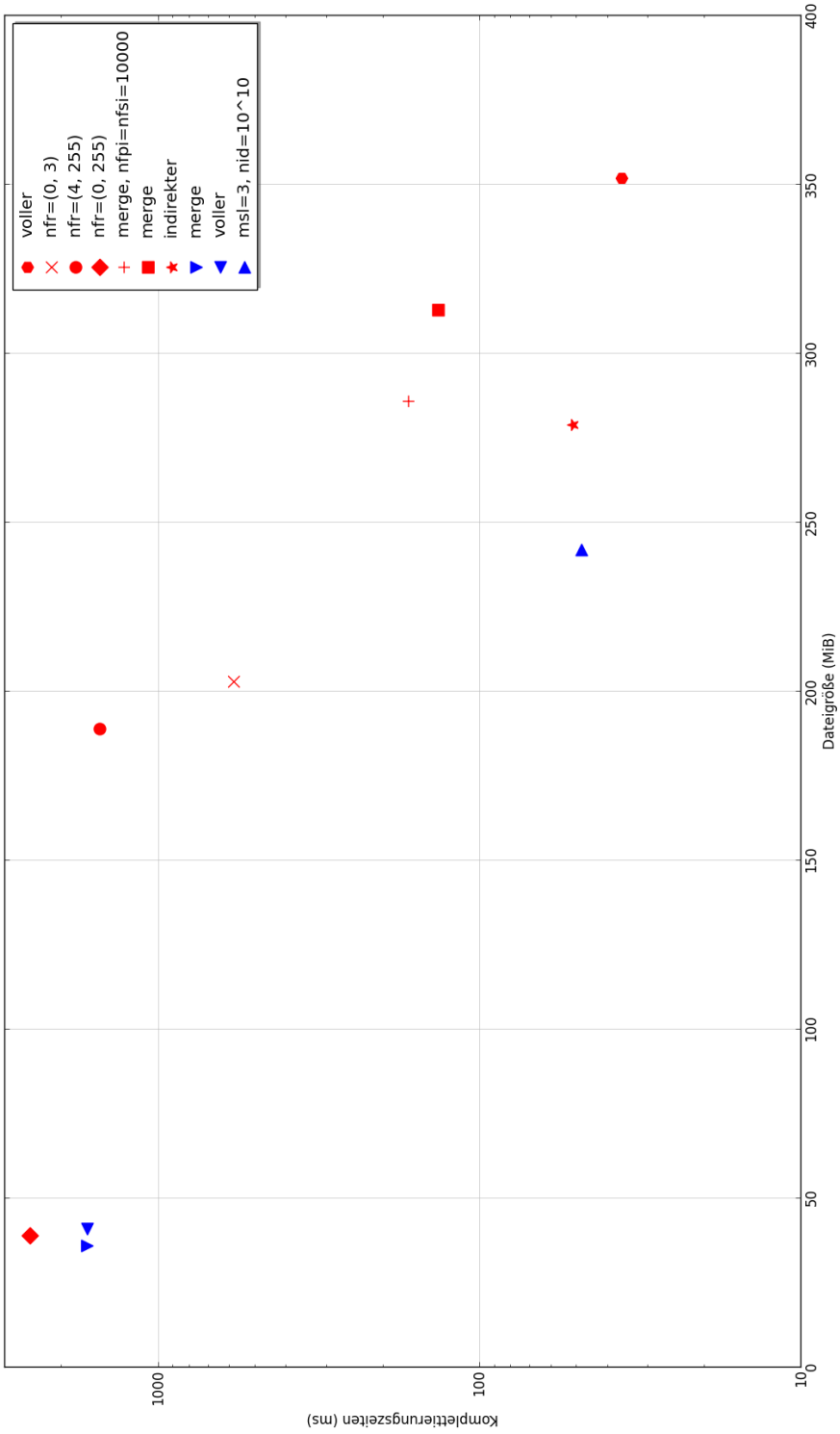


Abbildung 6.7: Baden-Württemberg-Datensatz: Mittlere Komplettierungszeiten ohne Mengenoperationen in Abhängigkeit von der Dateigröße des ItemIndexStores mit verschiedenen Komplettierern auf dem Mobiltelefon

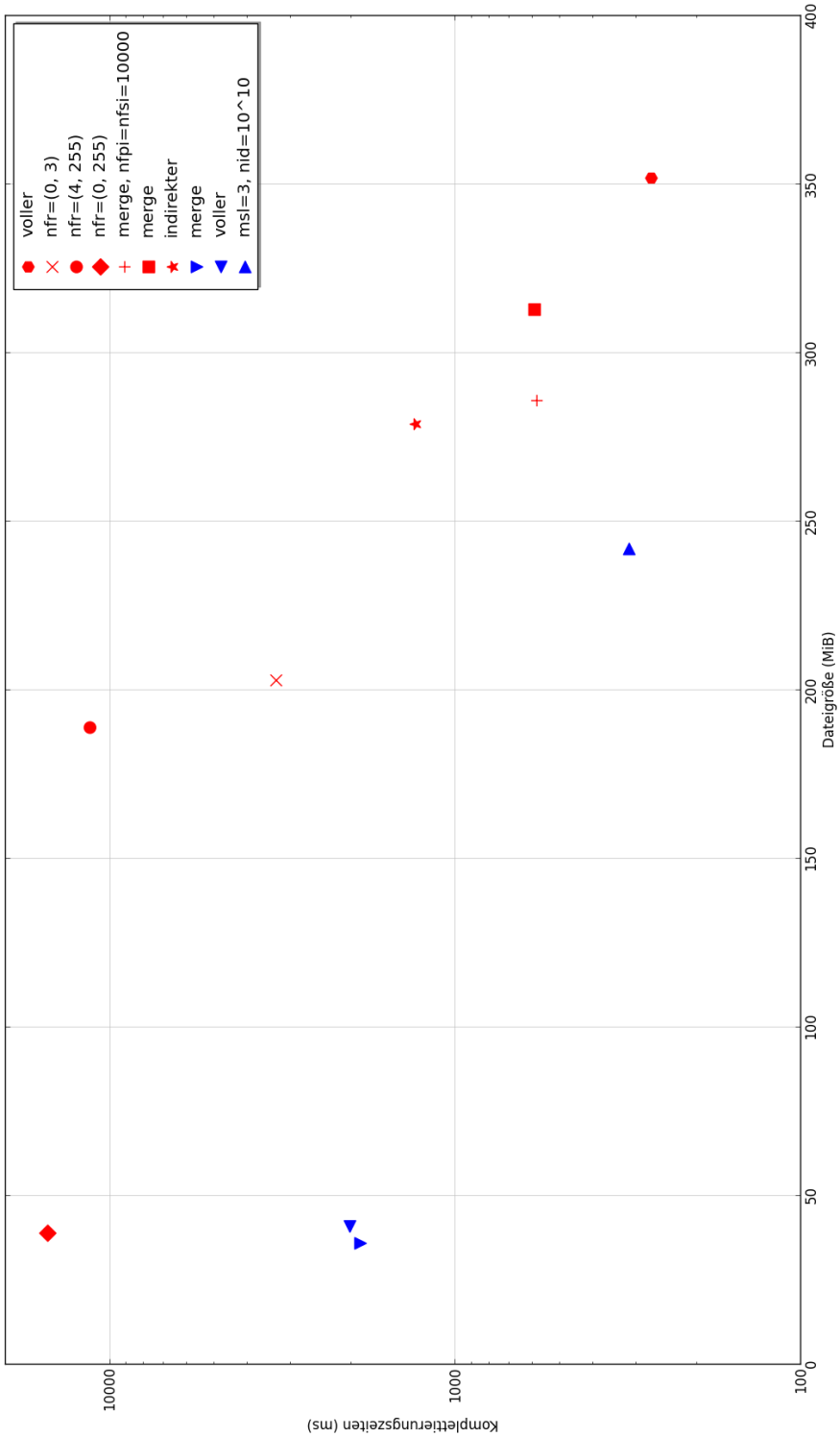


Abbildung 6.8: Baden-Württemberg-Datensatz: Mittlere Komplettierungszeiten mit Mengenoperationen in Abhängigkeit von der Dateigröße des ItemIndexStores mit verschiedenen Komplettierern auf dem Mobiltelefon

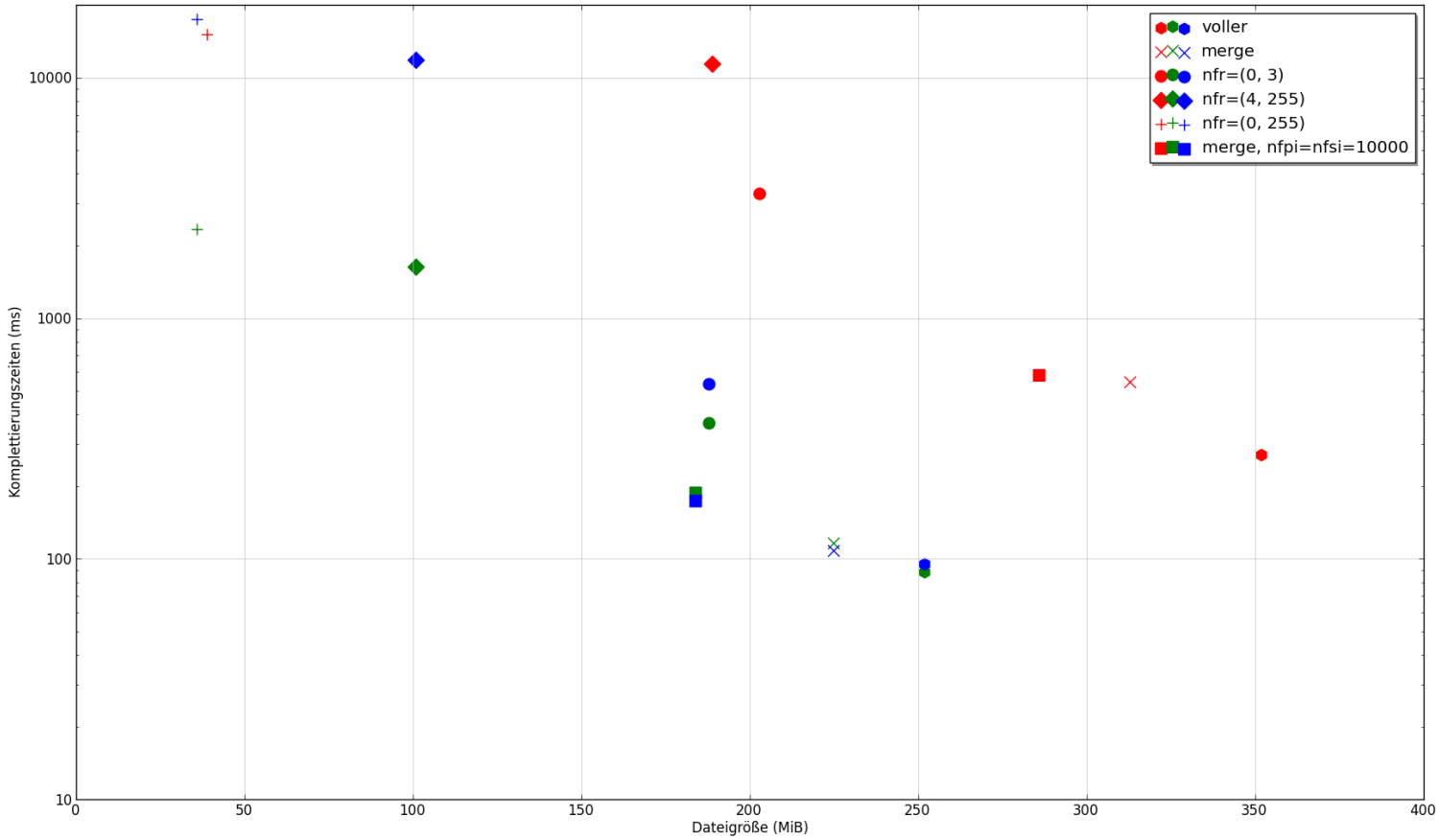


Abbildung 6.9: Baden-Württemberg-Datensatz: Vergleich der Komplettierungszeiten mit Mengenoperationen von ItemIndex-Speicherschemata mit Regressionsgerade (rot), Bit-Vektor (grün) und Differenzkodierung (blau) auf dem Mobiltelefon

	-s -adi	-s	-sd
Daten einlesen	16 s	16 s	16 s
Grenzpolygone schneiden	53 s	53 s	53 s
Trie erstellen	19 s	15 s	5 s
Trie mit vollem Index erstellen	36 s	32 s	5 s
Maximaler Speicherverbrauch	2,3 GiB	2 GiB	657 MiB
Knotenanzahl	2.308.044	1.907.143	286.770
Größe Trie	40 MiB	34 MiB	5,3 MiB
Größe ItemIndexStore	363 MiB	353 MiB	63 MiB
Komplettierungszeiten mit Mengenoperationen auf dem Mobiltelefon			
Minimum	62 ms	60 ms	4,2 ms
Maximum	647 ms	728 ms	607 ms
Mittelwert	265 ms	270 ms	133 ms
Standardabweichung	217 ms	233 ms	183 ms
Komplettierungszeiten ohne Mengenoperationen auf dem Mobiltelefon			
Minimum	0,3 ms	0,3 ms	0,3 ms
Maximum	335 ms	264 ms	109 ms
Mittelwert	43 ms	36 ms	92 ms
Standardabweichung	49 ms	32 ms	17 ms

Tabelle 6.19: Erstell- und Komplettierungszeiten für den Baden-Württemberg-Datensatz mit verschiedenen Suffix-Optionen

7 Zusammenfassung

In dieser Arbeit wurden verschiedene Ansätze zur Suche in OpenStreetMap-Daten auf dem Mobiltelefon aufgezeigt und evaluiert. Dabei hat sich gezeigt, dass der Generalisierte Suffix Trie (GST) für dieses spezielle Anwendungsszenario eine echte Alternative zum Suffix-Array-ähnlichen FlatGST darstellt. Gegenüber letzterem benötigt er etwas mehr Speicherplatz, lässt dafür komplexere ItemIndex-Speicherschemata zu. Darüberhinaus lassen sich zu gegebenem Knoten leicht die Kindknoten herausfinden, welche anschließend zur Ermittelung von nachfolgenden Zeichen der Sucheingabe genutzt werden können. Auch lässt sich ein GST mit Groß-/Kleinschreibung erstellen, auf welchem dennoch mit Missachtung der Groß-/Kleinschreibung gesucht werden kann. Auch ist die Suchzeit oft etwas kürzer als die des FlatGST, da die Anzahl der Kinder mit der Tiefe im Baum schnell abnimmt, wodurch die Suche nach dem passenden Kindknoten in den Hintergrund tritt und die Suche somit näherungsweise in linearer Zeit in Abhängigkeit von der Länge der Suchzeichenkette erfolgt. Demgegenüber hängt die Suchzeit im FlatGST hauptsächlich von der Anzahl der Knoten im GST ab. Betrachtet man den gesamten Speicherplatz zur Suche sowie die Zeit um Mengenoperationen durchzuführen, tritt die Geschwindigkeit der Komplettierer in den Hintergrund. Hier entscheidet vor allem der ItemIndex-Typ über die Größe der Daten sowie die Geschwindigkeit der Mengenoperationen. Hierfür wurden 3 unterschiedliche ItemIndex-Typen untersucht. Der Regressionsgeraden-basierte stellte sich als der Langsamste heraus, bietet gegenüber dem Bit-Vektor- und differenzkodierten ItemIndex jedoch wahlfreien Zugriff, was für den FlatGST mit Zeichenkettenreferenzen entscheidend ist. Der Bit-Vektor und differenzkodierte ItemIndex benötigte am wenigsten Speicherplatz und bot die schnellsten Komplettierungszeiten. Nur beim Vergleich von FlatGST mit Zeichenkettenreferenzen mit dem GST ohne Indices an inneren Knoten war der ItemIndex mit Regressionsgerade dem Bit-Vektor ItemIndex leicht überlegen. Darüberhinaus hat sich gezeigt, dass sich für bestimmte ItemIndex-Speicherschemata sehr hohe Kompressionsraten erreichen lassen, weshalb eine weitere Entwicklung in diese Richtung für sinnvoll erachtet werden kann. Die bisherige Implementierung bietet hierfür eine sehr gute Ausgangsbasis, da mit minimalem Aufwand neue Komplettierer und ItemIndex-Typen zum Projekt hinzugefügt werden können.

Literaturverzeichnis

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, SIGMOD '90, pp. 322–331. ACM, New York, NY, USA, 1990. doi:10.1145/93597.98741. URL http://doi.acm.org/10.1145/93597.98741. (Zitiert auf Seite 12)

[FB74] R. A. Finkel, J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974. (Zitiert auf Seite 12)

[geoa] Geofabrik Downloadbereich. URL http://download.geofabrik.de. (Zitiert auf Seite 16)

[geob] Geofabrik tools: Map Compare. URL http://tools.geofabrik.de/mc/. (Zitiert auf Seite 15)

[Gro12] Z.-O. Groß. Effiziente Darstellung von Kartendaten auf Mobilgeräten, 2012. (Zitiert auf Seite 16)

[Gut84] A. Guttman. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57, 1984. doi:10.1145/971697.602266. URL http://doi.acm.org/10.1145/971697.602266. (Zitiert auf Seite 12)

[icu12] ICU - International Components for Unicode, 2012. URL http://site.icu-project.org/. (Zitiert auf Seite 34)

[Kov] C. Kovalis. Long Range ZIP or Lzma RZIP. URL http://ck.kolivas.org/apps/lrzip. (Zitiert auf Seite 53)

[map] mapsforge - free mapping and navigation tools. URL http://code.google.com/p/mapsforge. (Zitiert auf Seite 44)

[MM90] U. Manber, G. Myers. Suffix arrays: a new method for on-line string searches. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, SODA '90, pp. 319–327. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1990. URL http://dl.acm.org/citation.cfm?id=320176.320218. (Zitiert auf Seite 11)

[nom] Nominatim. URL http://wiki.openstreetmap.org/wiki/Nominatim. (Zitiert auf Seite 13)

[Obe] M. F. Oberhumer. LZO real-time data compression library. URL http://www.oberhumer.com/opensource/lzo. (Zitiert auf Seite 53)

[ODb] Open Data Commons Open Database License (ODbL). URL http://opendatacommons.org/licenses/odbl/. (Zitiert auf Seite 15)

[osma] OpenStreetMap. URL http://www.openstreetmap.org/. (Zitiert auf Seite 15)

[osmb] Osmand (OSM Automated Navigation Directions). URL http://osmand.net. (Zitiert auf Seite 13)

[osmc] transparent map comparison openstreetmap google bing yahoo. URL http://sautter.com/map/. (Zitiert auf Seite 15)

[piz] Pizza and Chili Corpus - Compressed Indexes and their Testbeds. URL http://pizzachili.dcc.uchile.cl/index.html. (Zitiert auf Seite 12)

[qt-] Hauptseite des Qt Projekts. URL http://qt-project.org/. (Zitiert auf Seite 42)

[SRF87] T. K. Sellis, N. Roussopoulos, C. Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *Proceedings of the 13th International Conference on Very Large Data Bases*, VLDB '87, pp. 507–518. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987. URL http://dl.acm.org/citation.cfm?id=645914.671636. (Zitiert auf Seite 12)

[TA11] The Unicode Consortium, J. Allen. *The Unicode Standard, Version 6.0*. Sixth edition, 2011. URL http://www.unicode.org/versions/Unicode6.0.0/. (Zitiert auf Seite 34)

[Tri12] N. Trifunovic. UTF8-CPP: UTF-8 with C++ in a Portable Way, 2012. URL http://utfcpp.sourceforge.net. (Zitiert auf Seite 34)

[WOS06] K. Wu, E. J. Otoo, A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1):1–38, 2006. doi:10.1145/1132863.1132864. URL http://doi.acm.org/10.1145/1132863.1132864. (Zitiert auf Seite 36)

Alle URLs wurden zuletzt am 23.10.2012 geprüft.

Erklärung

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

(Daniel Bahrdt)