XML-Projekt SoSe 2012

GPSies.org, DBPedia.org, Twitter.com, XML/XSD/XSLT

Gruppe 11

Eike Cochu, Samer El-Safadi, Hannes Geist,
Cenk Gündogan, Michael Pluhatsch

15. Juli 2012

Inhaltsverzeichnis

1 Aufgabenstellung und Organisation

Gruppenleiter: Hannes Geist

1.1 Aufgabenstellung

Durch den XML-Endpoints von gpsies.com 110.000 Wanderstrecken laden, per XSLT-Schema in ein eigens entwickeltes XSD-Schema transformieren und in einer geeigneten XML-Datenbank speichern. Zu diesen Strecken soll per SPARQL der Endpoint von dbpedia.org abgefragt und alle auf (oder an) der Strecke liegenden Sehenswürdigkeiten abgefragt werden können. Ein HTML-Formular (oder ähnliches) entwickeln, mit dem die XML-Datenbank abgefragt werden kann. Das Ergebnis der Abfrage soll wiederum per XSLT in HTML transformiert und zusammen mit den vorhandenen Sehenswürdigkeiten und den zu diesen Sehenswürdigkeiten vorhandenen Tweets von Twitter.com angezeigt werden.

1.2 Organisatorisches

Zur Gruppenkoordinierung hat sich unsere Gruppe jede Woche Sonntags beim Gruppenleiter Hannes Geist getroffen und das weitere Vorgehen besprochen und gemeinsam an den einzelnen Softwarekomponenten gearbeitet. Diese Dokumentation ist gemeinsam entstanden, wobei jeder seinen Aufgabenteil beschrieben und etwaige Probleme, die im Verlauf der Bearbeitung auftraten, aufgezählt und ausgeführt hat.

Die endgültigen Ergebnisse sind momentan noch unter erreichbar.

1.3 Aufgabenverteilung und -bewältigung

- Eike Cochu: Erstellung des XSD-Schemas, Bereitstellung eines Servers, Dokumentation
- Samer El-Safadi: SPARQL-Abfrage der Sehenswürdigkeiten von dbpedia.org
- Hannes Geist: XSLT-Transformierungen und Füllen der XML-Datenbank
- Cenk Gündogan: Abfrage der Strecken von gpsies.org
- Michael Pluhatsch: HTML-Formular mit PHP, Abfrage der XML-Datenbank

1.3.1 Eike Cochu

Aufgabenbereich: Erstellung eines XSD-Schemas für die XML-Datenbank, Bereitstellung des Datenbank- und Applikationsservers sowie Installation der Datenbank, Einfügen der Inhalte in

die Datenbank und Installation der Webseite, kleine kosmetische Änderungen an der Webseite und Formatierung, Erstellen der Dokumentation.

XSD-Schema Nachdem wir uns mit einem kurzen Testlauf einige Anschauungsdaten von gpsies.org beschafft hatten, konnten wir basierend darauf grob die Anforderungen an das XSD-Schema entwerfen. Die Idee des Schemas sollte es sein, mehr mit Attributen zu arbeiten und so die Anzahl an Elementen zu verringern, damit die Speichergröße der resultierenden Dateien minimal gehalten wird. Dazu haben wir uns entschlossen, einige weniger wichtige Datenteile der crawl-Daten garnicht erst mit in die Datenbank zu speichern, im Schema waren diese dann auch nicht vorgesehen. Die meisten Elemente sind mittels minOccurs=0 ebenfalls optional gehalten, Attribute sind per default optional.

Zur Veranschauung ein kleiner Ausschnitt aus dem Schema:

```
<!-- Eine Adresse. Optional: alles -->
<xsd:complexType name="address">
  <xsd:attribute name="street" type="xsd:string"/>
  <xsd:attribute name="streetnumber" type="xsd:string"/>
  <xsd:attribute name="zipcode" type="xsd:string"/>
  <xsd:attribute name="city" type="xsd:string"/>
  <xsd:attribute name="country" type="xsd:string"/>
</xsd:complexType>
```

Hier ist zu sehen, dass in unserem Schema hauptsächlich Attribute verwendet werden, um die Anzahl an Elementen zu verringern. Auch sind Attribute standardmäßig optional, was bei Elementen erst hätte eingerichtet werden müssen.

Beim Schema sind wir auf keine Probleme gestoßen. Das XSD erlaubt es auf einfache Art und Weise ein sehr dynamisches Schema zu erstellen, dass wir im Prozess der Entwicklung aufgrund von neuen Erkenntnissen immer wieder verädert haben, bis es allen Anforderungen entsprach. Die Validierung des Schemas wurde mit einem gewöhnlichen XSD-Schemavalidierer durchgeführt, der unser Schema mit dem eigentlichen XSD-Schema validieren konnte.

Datenbank Wir haben uns für BaseX als Datenbank entschieden, da dieses auf Java basiert und somit systemunabhängig installierbar und auch leicht zu handhaben ist. Die Datenbank wurde auf einem dedizierten Ubuntu 12.04 Server installiert, als Webserver wurde Apache + PHP 5 gewählt.

Probleme mit der Datenbank: Da das entgültige Datenbankfile eine Größe von 1.2 GB hatte und der Heapspeicher der JVM default sehr klein ist, trat beim Erstellen der Datenbank immer

ein Out Of Memory Fehler auf, den ich erst mit dem manuellen hochsetzen des Heapspeichers beheben konnte (`java -cp basex.jar org.basex.BaseXServer -Xmx1G`). Bei genauerer Untersuchung habe ich herausgefunden, dass dieser Speicherfehler speziell durch die Volltextindizierung ausgelöst wird, die auch gezieht ausgeschaltet werden kann.

1.3.2 Samer El-Safadi

Bei der Aufgabe die XML-Datenbank um relevante „Points of interest"-Daten zu erweitern, sind wir uns sehr schnell einig gewesen, uns auf Deutschland zu konzentrieren und sämtliche andere Länder außer Acht zu lassen.

Um an jene Punkte zu kommen, mussten wir uns erst einmal über die Abfragesprache SPARQL informieren. Dies haben wir größtenteils über das Buch „Learning SPARQL" von Bob DuCharme getan, in dem sowohl allgemeine Informationen über die Materie (sprich das „Semantic Web", RDF, Linked Data etc.), als auch spezielle Situationen dargestellt sind, die über SPAQL-Abfragen anschaulich bearbeitet werden. Praktischerweise beziehen sich einige der Beispiele direkt auf DBpedia, wodurch wir das Geschriebene auch gleich testen konnten.

Nachdem wir damit nun eigene Abfragekonstrukte erstellen konnten, die wir über den Online-Zugriff ausprobiert haben, ging es nun darum den SPARQL-Endpoint nun auch über Java ansprechen zu können. Schnell fiel die Wahl auf das Jena Framework von Apache, welches uns einen sehr komfortablen Weg ermöglichte über ARQ auf den entsprechenden Endpunkt zuzugreifen: Wir konnten also durch die QueryFactory entsprechende Anfragen erstellen und diese über QueryExecutionFactory.sparqlService an einen definierten Service (in diesem Fall DBpedia) senden.

Das Programm machte uns durch Warnungen darauf aufmerksam, dass kein Logger eingerichtet worden ist. Wir behoben dies gleich und integrierten log4j (ebenfalls Apache), um die Ergebnisse der Anfrage in ein Logfile zu schreiben, statt es nur (testweise) in die Standardausgabe zu packen. Als dies erledigt war, konnten wir über die Ergebnisse (ResultSet) iterieren (.hasNext() bzw. .next()) und diese dann über den Logger direkt in eine Datei schreiben.

Da das Ergebnis allerdings ein XML-Dokument sein sollte, um es über XSLT entsprechend einfach umzuwandeln, mussten wir erstmal eine Art Parser schreiben, der über die Iteration der Ergebnisse ein wohl geformtes XML-Dokument erstellt.

Dies funktionierte zwar, stellte sich allerdings als sehr aufwendig heraus, so dass wir uns nach

einer neuen Lösung umgesehen haben: Jenas ResultSetFormatter.outputAsXML war die perfekte Lösung unserer Probleme.

Wir mussten entsprechend umdisponieren, haben den log4j-Logger wieder entfernt und entsprechend mit dem Fileoutputstream gearbeitet, den wir komfortabel über den ResultSetFormatter von Jena dazu bringen konnten uns die Ergebnisse in eine sehr übersichtliche XML-Datei zu parsen.

Nun stand das Programm und es ging darum unsere Abfragekonstrukte durch sinnvolle Abfragen zu ersetzen. Dafür mussten wir viel Zeit investieren, um uns erst einmal einen Überblick zu verschaffen, wie die Datensätze in der DBpedia angeordnet waren. Wir merkten schnell, dass die englische DBpedia weitaus weniger relevante Daten über Deutschland enthielt, als die deutsche - die allerdings wiederum weitaus weniger GEO-Datensätze über die entsprechenden Objekte enthielt (vergleiche http://de.dbpedia.org/page/Brandenburger_Tor
mit http://dbpedia.org/page/Brandenburg_Gate).

Außerdem mussten wir feststellen, dass viele Objekte oftmals falsch bzw. scheinbar keiner Regel folgend in entsprechende Typen eingeordnet wurden und alles doch recht chaotisch war, sodass wir nicht einfach eine große Anfrage starten konnten und die Ergebnisse anschließend anhand ihrer Typen zuordnen konnten. Wir verbrachten viel Zeit damit die yago-Sprachklasse bzw. die Kategorien von DBpedia zu studieren, um entsprechend gezielte Anfragen zu stellen. Mindestvoraussetzung eines Objektes war für uns die Bezeichnung, Geo-Daten in Form von long und lat sowie ein Link zum entsprechenden Wikipedia-Artikel, damit der Nutzer sich weitere Informationen einholen konnte.

Durch dieses Vorgehen waren wir in der Lage Anfragen zu entwickeln, die relativ schnell abgearbeitet wurden und wir wussten genau, um welche Untermenge es sich bei den Ergebnissen handelte, sodass wir dies entsprechend in der Datenbank vermerken konnten. Dadurch konnten wir die Website entsprechend so einrichten, dass der Nutzer es sich einzeln aussuchen konnte, ob dieser Parks, Seen etc. angezeigt bekommen wollte oder nicht.

1.3.3 Hannes Geist

Aufgabenbereich Erstellung des DB-XML-Dokuments aus den Daten des GPSies Crawlers per XSLT und Zusammenführung mit den Daten des POI-Crawlers per Java, Nutzung von SAX

und StAX.

Das XSLT-Dokument Die Ausgangsdaten bestehen aus dem Document-Element „crawl“ und dem einzigen Kindelement „query“, welches dann jeweils einen „track“-Knoten als Ergebnis der GPSsies-Anfragen enthält. Die Informationen in einem Trackknoten liegen in einer flachen Hierarchie vor, es existieren keine Attribute und Kindknoten sind maximal noch einmal verschachtelt um bestimmte mehrere Werte einer bestimmten Track-Eigenschaft aufzuzählen. Das Tranformationsdokument besteht aus einer einfachen Template zur Erfassung jedes track-Knotens im GPSies-Crawl und einem komplexeren Template zur Transformation eines solchen. Dieses komplexere track-Knoten-Template fügt dem zu schreibenden track-Knoten im neuen Dokument je nach Vorhandensein des Knotens in einem track-Knoten des Ausgangsdokuments die Attribute trackName, author, createTimestamp, totalLength, totalAscend, totalDescend, altitudeMaxHeightM, altitudeMinHeightM, altitudeDifferenceM, totalDescendM und quality, sowie die Kindelemente fileId, kmlLink, trackProperty, description, trackAttributes, trackCharacters, trackRoadbeds, trackRoads, trackTypes und points hinzu. Dem points-Knoten werden point-Knoten als Kindelemente zugeordnet, die aus der Aufzählung der GPS-Positions- und Höhenkoordinaten extraiert werden und die Attribute lon für die geographische Länge, lat für die geographische Breite und ele für die Höhe über Normalnull erhält.

Der GPSies-POI-Parser Es handelt sich um ein Java-Programm, das zur Ausführung als Eingabedateien die in der main-Methode im Feld poiFiles festgelegten XML-Dokumente, sowie die aus der oben beschriebenen XSL Transformation resultierende XML-Datei benötigt. Die main-Methode befindet sich in der Klasse „Parser“.

Programmablauf Zuerst werden die aufgeführten POI-Dokumente über die Methode parsePois der Klasse PoiParser in DOM-Objekte geparst, die einer Hashmap „pois“ mit dem Dateinamen als Schlüssel und ArrayListen von Poi-Objekten als Werten besteht. Die Klasse Poi bildet einen POI mit den Eigenschaften Titel(title), den Wikipedia-Link(wikiLink) und die GPS-Koordinaten(lon, lat), ohne Höhenangabe, eines POIs ab. Diese Vorgehensweise ist nur für eine relativ kleine Anzahl von POIs bzw. POI-Dokumenten sinnvoll, wie im aktuellen Stand unserer Software. In einem zweiten Schritt wird nun die Methode connectPois eines Objekts der Parser-Klasse aufgerufen, die einem SAX-Parser-Objekt eine Instanz der Klasse PoiHandler übergibt, in der die eigentliche Zuordnungslogik von POIs zur GPSies-Datenbankdatei implementiert ist. Der PoiHandler erweitert den DefaultHandler des SAX-Parsers so, dass ein XML-Dokument mit dem Namen „gpsies_pois.xml“ erzeugt wird. Das Document-Element heißt „tracks“ und wird

wie alle anderen Elemente des Ausgangsdokuments in der Handler-Methode „endElement“ automatisch in das neue Ausgabedokument geschrieben. Eine Ausnahme bilden die track-Elemente, die in der genannten Methode eine Sonderbahndlung erfahren. Deren points-Kindelemente mit den enthaltenen point-Elementen werden ausgelesen und ein Rechteck aus dem am weitesten südlich/westlich und dem am weitesten nördlich/östlich gelegenen Punkt + 10% der errechneten Rechteckhöhe bzw. -breite errechnet, ausgehend vom, bezüglich Gesamtzahl der Punkte, in der Mitte der Trackpunktmenge liegenden Trackpunkt.

Folgende Anmerkungen zur eingereichten Version: Die aktuelle Softwareversion enthält noch einen Test bzw. einen Fehler in der Methode connectPois, bei dem unter dem Namen der zu erzeugenden Ausgabedatei ein XML-Writer- und ein Dateistrom erstellt werden und mit einem leeren Element beschrieben werden. Im PoiHandler werden XML-Writer und Dateistrom am Ende des Parsevorgangs nicht explizit geschlossen, was im aktuellen Fall keine Fehler erzeugt, jedoch bei einer Weiterentwicklung zu Fehlern führen könnte.

1.3.4 Cenk Gündogan

Aufgabenbereich: Entwerfen und Implementieren eines Crawlers, der mind. 100.000 Tracks vom gpsies.org Server herunterlädt und lokal in einer gültigen XML-Datei abspeichert.

Bei der Entwicklung des Crawlers wurden die Eigenschaften „Stabilität“ und „Schnelligkeit“ sehr hoch priorisiert. „Stabilität“ bedeutet, dass der Crawler bei Verbindungsabbrüchen weiter arbeiten kann und als Resultat immer noch eine korrekte XML-Datei mit gültigem Inhalt produziert wird. „Schnelligkeit“ des Crawlers wird durch Parallelisierung des Verbindungsaufbaus zum Server gewährleistet. Eine weitere nennenswerte Eigenschaft ist die Tatsache, dass das Resultat des Crawlers in komprimierter Form (mit gzip) erstellt wird. Das Ergebnis ist dann sehr einfach und unproblematisch auf verschiedene Rechner übertragbar.

Die Anfragen an den Server geschehen via Http POST und sehen folgendermaßen aus:

key	der API-Key wurde von gpsies.org bereitgestellt. Ohne diesen ist eine Anfrage an den Server nicht möglich
country	wir beschränken uns nur auf Tracks aus Deutschland
filetype	Als Dateityp der GPS-Daten erwarten wir KML
limit	es sollen **limit** viele Track-Ids übertragen werden
resultpage	pro Query können immer nur 100 Tracks übermittelt werden, mit inkrementierender **resultpage** kann man die nächsten 100 Tracks abfragen

1. URL1: *http://www.gpsies.org/api.do?**key**=<key>&**country**=DE& **limit**=100&**resultpage**=i*
   Es werden die **i.**en 100 Tracks geladen
2. URL2: *http://www.gpsies.org/api.do?**key**=<key>&**limit**=100 &**filetype**=kml&**fileId**=f1&**fileId**=f2...*
   Die 100 TrackIds, die mit URL1 erfragt wurden, werden nun alle an die 2. URL angefügt. Mit URL2 erhalten wir detaillierte Informationen zu jedem Track

Zur Realisierung des Crawlers wurde ein Maven-Projekt erstellt und zur leichten Handhabung der REST-Abfragen der HttpClient von org.apache.httpcomponents benutzt. Die Programmiersprache ist Java.

Der Crawler besteht aus zwei Hauptkomponenten, dem Master und die Worker. Zuerst wird vom Master ein komprimierter Output-Stream(gzip), verknüpft mit einer lokalen Datei, geöffnet. Der Master stellt nun in einer Schleife, die von **i=0** bis **i=*iterations*** geht, die Anfrage (URL1) an den Server, um die **i.**en 100 TrackIds zu erhalten. Aus der Response des Servers werden mittels regular expression die einzelnen TrackIds heraus gefiltert. Mit den erhaltenen TrackIds kann nun die zweite Abfrage an den Server gestellt werden (URL2), um detaillierte Informationen zu jedem Track zu erhalten. Diese zweite Response wird nun wieder mittels regular expression nach Tracks untersucht. Bei jedem Fund eines Tracks, wird ein Worker erzeugt und diesem der gefundene Track übergeben. Der Master wiederholt diesen Prozess bis keine weiteren Tracks in der Response gefunden werden und wartet auf die Terminierung aller Worker. Wenn alle Worker terminiert sind, entsteht ein gültiger XML-Abschnitt, der in den komprimierten Output-Stream geschrieben und geflusht werden kann. Nun beginnt die nächste Iteration des Masters und dieser Prozess führt sich fort, bis **i=iterations** erreicht ist.

Wird ein Worker erstellt und ihm ein Track übergeben, wird der Download-Link der KML-Datei dieses Tracks via regular expressions heraus gefiltert. Die KML-Datei wird mit einem InputStreamReader geöffnet und die enthaltenen Koordinaten ausgelesen. Wir entschieden uns hierbei für eine „nimm jede 10.“-Politik, da es überaus viele, sehr dicht beieinander liegende, Koordinaten zur Verfügung gab. Zum Beenden des Workers wird der Track samt ausgelesener Koordinaten an den Master übergeben.

Das Resultat ist eine komprimierte XML-Datei mit 100.000 Tracks + Koordinaten. Die Größe der Datei liegt bei ungefähr ∼200MB. Entpackt liegt die Größe bei ∼1.1GB. Die ideale Crawl-Dauer liegt bei ∼3.5h-4h

Aufruf des Crawlers: `java -jar ./crawler ,,api-key‘‘ ,,/path/to/file.xml.gz‘‘ ,,number of iterations‘‘`
Die Dateiendung muss „gz“ sein, da eine komprimierte Datei erzeugt wird. Eine Iteration beinhaltet im besten Fall 100 Tracks, so erhält man bei 1000 Iterationen 100.000 Tracks.

Probleme: Das schwerwiegenste Problem war die Tatsache, dass der Crawler aufgrund von langen Ausfallzeiten seitens des Servers nicht genügend Tracks produzieren konnte und durch viele Timeouts um mehrere Stunden verlangsamt wurde.

1.3.5 Michael Pluhatsch

2 Abschließende Präsentation