Fachhochschule Brandenburg
University of Applied Sciences

Systemintegration

Projekt Scrapedia

Hoffmann, Benjamin	20090020	hoffmanb@fh-brandenburg.de
Mosters, Curtis	20090033	mosters@fh-brandenburg.de
Mögelin, Josef	20090028	moegelin@fh-brandenburg.de
Seetge, Mathias	20090035	seetge@fh-brandenburg.de
Lange, Stefan	20090009	langeste@fh-brandenburg.de

Dozent:	Prof. Dr.-Ing. T. Preuß Dipl.-Inf. Mark Rambow, M.Sc.
Lehrveranstaltung:	Systemintegration
Abgabedatum:	23.01.2013
GitHub:	https://github.com/sicLotus/Scrapedia

Version 1.0 *23.01.2013*

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Das World-Wide-Web besteht aus einer enormen Menge von verknüpften HTML-Dateien. Internetseiten und Verbindungen zwischen ihnen können als Knoten und gerichtete Kanten eines Graphs aufgefasst werden. Es gibt viele Versuche, das Internet zu visualisieren. [Kumar, 2000]

In dieser Arbeit wollen wir ein Projekt realisieren, welche Aspekte des Webcrawlings und verteilter Systeme miteinander verknüpft, und unter Einsatz neuer, aufstrebender Technologien am Beispiel des deutschsprachigen Wikipedias einen solchen Graphen erstellt und statistische Auswertungen ermöglicht.

Zudem bietet sich hierbei eine tolle Gelegenheit, unser theoretisches Wissen zur Systemintegration und zum weiten Gebiet der verteilten Systeme in einem interessanten Projekt praktisch einzusetzen.

1.2 Warum Wikipedia

Wikipedia ist das derzeit meistgenutzte Online-Nachschlagewerk weltweit. Es stellt eine riesige Menge an realen Daten zur Verfügung, die die Verwendung von gut skalierbarer Technologie notwendig macht. Darüber hinaus bieten die Server von Wikipedia eine sehr hohe Verfügbarkeit und Performanz, welches das Arbeiten damit erheblich verbessert. Darüber hinaus ist es sehr interessant Rückschlüsse auf Artikel und Beziehungen zwischen diesen ziehen zu können. Daher ist es interessant einen Überblick über die gesammelten Daten dieses Online-Dienstes zu erhalten.

Im Folgenden wird unter dem Begriff Wikipedia lediglich der Bezug zum deutschen Wikipedia getroffen.

1.3 Zielsetzung / Projektvorstellung

Scrapedia ist die prototypische Entwicklung eines verteilten Web-Scrapers zur Extraktion ausgehender interner Wikipedia-Links auf Basis von AWS-Technologien und Nutzung dieser als Verknüpfungselemente zur Generierung einer Artikelnetzstruktur.

Jeder Artikel im deutschen Wikipedia wird gescrapt. Dabei werden sämtliche interne Links in einer Datenbank gespeichert.

Die Auswertung der gesammelten Daten soll in folgenden Varianten erfolgen:

- Grafische Darstellung einer Auswahl an Knoten/Kanten
- Umsetzung einer Pfadsuche (kürzester Weg von Artikel A zu Artikel B)
- Generelle Aussage über die Erreichbarkeit von Artikeln (Zum Beispiel: „Wie viele Artikel verlinken auf den Artikel ‚Philosophie'?", sonstige Statistiken (Top 10))

In dieser Arbeit wird die Herangehensweise und Durchführung des Projektes dokumentiert und erläutert. Dabei werden mögliche Ansätze zur Lösung vorgestellt, sowie die einzelnen Schritte zur Lösung der Aufgabe erklärt. Es werden die verwendeten Technologien vorgestellt und zukünftige Schritte dargelegt. Abschließend wird auf Probleme und Bottlenecks während der Realisierung eingegangen.

2 Das Projekt

2.1 Herangehensweise und mögliche Ansätze

2.1.1 Vorüberlegung

Der Ansatz eines klassischen Webcrawlers, bei einer beliebigen Seite mit der Analyse zu beginnen, dessen Informationen zu speichern und auf Basis der gefundenen Hyperlinks die nächsten Prozessschritte und damit abzusuchenden Artikel zu finden, stellt sich bei näherer Betrachtung nicht als effektivste Methode heraus. Zwar kann so ein für den Menschen leicht nachvollziehbares System verwendet werden, jedoch ist weder die Anzahl der nötigen Prozessschritte abschätzbar, noch kann die Vollständigkeit der Analyse gewährleistet werden. So können beispielsweise Artikel ohne eingehende Verlinkungen nicht erreicht werden.

Um eine Wikipedia-Instanz zu analysieren, wird daher im Idealfall eine Liste aller ihrer Artikel verwendet. Mit Hilfe dieser Liste kann jeder Eintrag gescrapt und nach internen Verlinkungen durchsucht werden. Die Ergebnisse werden anschließend gespeichert und können daraufhin ausgewertet

werden. Diese Basiskonzeption, welche in Abbildung 1 skizziert ist, soll als Leitfaden der Entwicklung dienen.

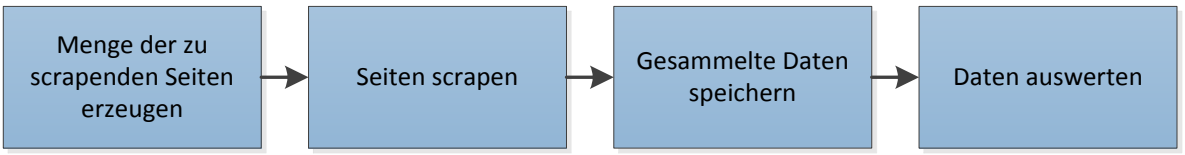

Abbildung 1: Grober Ablauf

2.1.2 Liste aller Wikipedia-Artikel

Um eine Liste aller Wikipedia-Artikel zu erhalten, wurden drei Möglichkeiten betrachtet:

1. Der komplette Artikelbestand Wikipedias wird im Spezial-Artikel „Von A bis Z“ [1] aufgelistet, allerdings sind diese auf mehrere Unterseiten verteilt. Um eine Liste sämtlicher Artikel zu erhalten, müsste ein Skript alle Seiten iterativ/rekursiv durchlaufen und die Hyperlinks der Artikel sichern. Diese Variante ist zeitaufwändig und relativ umständlich.
2. Über die Wikipedia API[2] können einzelne Artikelinformationen (wie Titel, PageID, URL) mit Hilfe eines Queries abgerufen werden. Allerdings ist der Rückgabewert je Anfrage auf maximal 500 Einträge beschränkt. Die Darstellung kann in verschiedenen Formaten (XML, JSON) erfolgen. Da Wikipedia ungefähr 2,5 Millionen Artikel besitzt, müssten daher ca. 5000 Queries an die API gesendet und die Ergebnisse gespeichert werden. Diese Variante wurde zuerst implementiert jedoch im späteren Verlauf aufgrund von Performanceproblemen verworfen.
3. Wikipedia stellt eine Datei sämtlicher Artikel über ein Online-Verzeichnis[3] zur Verfügung. Diese Datei kann heruntergeladen (Größe: ca. 14 MB) und entpackt werden. Sämtliche Artikeltitel sind in ihr unformatiert aufgelistet. Die Analyse dieser Datei ist bei weitem die performanteste Lösung und benötigt je nach Internetverbindung zwischen 1-2 Minuten. Ein Nachteil dieser Methode ist, dass Wikipedia die Liste aller Artikel erst mit Verzögerung aktualisiert. Dies hat zur Folge, dass Artikel in der Liste eventuell noch nicht auftauchen oder nicht mehr auftauchen sollten, da sie bereits gelöscht wurden. Hierbei handelt es sich jedoch um eine sehr geringe Anzahl an Titeln und dies kann damit vernachlässigt werden.

2.1.3 Speicherung der Artikel und seiner Verknüpfungen

Für die Speicherung eines Artikels und die Verlinkung untereinander bietet sich die Verwendung einer Graphendatenbank an. Jeder Artikel (Knoten) besitzt zwei Eigenschaften: URL und Titel. Die URL

[1] http://de.wikipedia.org/wiki/Spezial:Alle_Seiten
[2] http://de.wikipedia.org/w/api.php
[3] http://dumps.wikimedia.org/dewiki/latest/

wird als Index zur eindeutigen Identifizierung verwendet. Artikel, die lediglich Weiterleitungen zu andere Artikel sind, werden hierbei ebenfalls erfasst.

Der Vorteil einer solchen Datenbank liegt in ihrer Struktur. Die Suche in einem Graphen, beispielsweise nach dem kürzesten Pfad zwischen zwei Knoten, ist hier deutlich performanter realisierbar und muss nicht umständlich implementiert werden.

2.1.4 Scrapen eines Artikels

Beim Web-Scraping handelt es sich um eine Technik, um Informationen von einer Webseite zu extrahieren. Die Wikipedia-Artikel werden nach Hyperlinks auf andere interne Seiten durchsucht und gespeichert. Hierbei müssen externe Links, Sprungmarken-Verlinkungen und spezielle Wikipedia-Steuerungslinks ausgefiltert werden.

2.1.5 Verteilung

Aufgrund der Größe des Datenbestands müssen einzelne Aufgaben verteilt werden, um den gesamten Ablauf in einer annehmbaren Zeit durchzuführen. Die Anwendung sollte daher skalierbar sein. Nach Analyse der einzelnen Prozesse stellte sich nur das Scrapen eines Artikels als sinnvoll und problemlos zu verteilen heraus. Dieser Schritt beinhaltet sowohl das Verteilen auf mehrere Rechner, als auch das mehrfache Ausführen desselben Skripts auf einer Instanz.

2.1.6 Auswertung

Für die Auswertung des Projektes sollen folgende Statistiken generiert werden:

- Top 20 der häufigsten Verlinkungen
- Top 20 der Artikel mit den meisten Links
- Liste aller nicht verlinkten Artikel
- Liste aller Artikel ohne Verlinkungen

Darüber hinaus wird eine Pfadsuche angeboten, die die kürzesten Pfade zwischen zwei Artikeln sucht und grafisch darstellt.

2.2 Technologien

2.2.1 Neo4j

Neo4j ist eine in Java implementierte Open-Source-Graphendatenbank und wurde von Neo Technology entwickelt. Sie ist hoch skalierbar, zuverlässig dank vollständigen ACID Transaktionen und ermöglicht eine schnelle Durchsuchung der Graphen. Darüber hinaus verwendet sie eine eigene Query-Language (Cypher), die leicht verständlich und schnell erlernbar ist. Eine Übersicht von weiteren Gründen, welche für den Einsatz von Neo4j sprechen befinden sich in [Hoff, 2009].

Die wohl bekannteste Graphendatenbank Neo4j wurde für dieses Projekt gewählt. Neo4j kann in Verbindung mit Java, PHP, Ruby oder Python verwendet werden. Dabei werden zwischen zwei Schnittstellen unterschieden: RESTful und Embedded. Beide Möglichkeiten wurden implementiert und miteinander verglichen. Die REST-Schnittstelle ist deutlich langsamer[4] als die Embedded-Variante. Für die riesigen Mengen an zu verarbeitenden Daten sollte zu mindestens schreibend nicht über REST zugegriffen werden.

Die horizontale Skalierung von Graphendatenbanken stellt ein großes Problem dar. Es besteht bei Neo4j die Möglichkeit einer Skalierung, wobei jedoch nur die Performance von lesenden Zugriffen verbessert werden kann. [Montag, 2012] Eine Partitionierung der Daten müsste manuell erfolgen und bietet sich bei einem einzigen, vernetzten Graphen nicht unbedingt an.[5] [Webber, 2011]

2.2.2 Python

Aufgrund von Performance-Schwierigkeiten muss eine Applikation mit Embedded-Zugriff auf dem Datenbankserver eingesetzt werden. Aufgrund der Schnittstellen von Neo4j bestand die Wahl zwischen Java und Python. Es wurde sich für Python entschieden, welche die typischen Vorteile von Skriptsprachen (einfach zu benutzen, lose Struktur) aufweist und somit ideal für die schnelle Umsetzung des Datenbank-Zugriffs ist.

[4] Schreibperformance: 25 Artikel mit Links/s bei Embedded im Gegensatz zu 1 Artikel mit Links/s via REST-Schnittstelle

[5] Weitere Informationen über Verteilungsansätze mittels Neo4j stehen unter http://jim.webber.name/2011/03/strategies-for-scaling-neo4j/ zur Verfügung.

2.2.3 Redis

Zur Verteilung der Aufgaben wird Redis, ein Key-Value-Store, verwendet. Die Liste aller Artikel wird darin gespeichert und an andere Instanzen verteilt. Um die Konfiguration der Worker zu entkoppeln, werden auch Einstellungen wie zum Beispiel der Key-Name der jeweiligen Redis-Liste und die URL bzw. der Port des Datenbankservers in Redis gespeichert und von den abarbeitenden Skripten verwendet. Redis-Listen dienen als Queues für die zu scrapenden Titel und das Ergebnis der gescrapten Artikel (Artikel mit Links im XML-Format). Dies wird in Abbildung 2 veranschaulicht.

Da wir aus Performance-Gründen eine Embedded-Anwendung zur Speicherung der Daten in der Datenbank verwenden, lässt sich dieser Teil nicht verteilen, denn die Anwendung muss auf dem Datenbankserver liegen. Würde man jedoch die Datenbank verteilen, so könnten ebenfalls die Applikationen zur Speicherung der Ergebnisse verteilt werden.

Redis besitzt die Möglichkeit der Replikation. Das bedeutet, dass Änderungen auf dem Master auf die Slaves übertragen wird. Dies ermöglicht eine Skalierung der Lesezugriffe und der Datenredundanz. [Red13]

Der große Vorteil von Redis ist die sehr einfache Einbindung in Anwendungen und die damit entstehende Entkopplung einzelner Komponenten. Des Weiteren ist Redis extrem schnell, da Daten im Hauptspeicher gehalten werden und kann somit ca. 100.000 (einfache) Schreib- und über 80.000 Lesevorgänge erreichen. [Redis] Einschränkungen bestehen bei der Verschachtelung von Datentypen und der Abfrage von einzelnen Werten.

Alternative Ansätze wie zum Beispiel der Einsatz von SQS oder Memcached (anstelle von Redis) und MapReduce zum Scrapen der Daten wurden betrachtet, aber nicht für besser befunden.

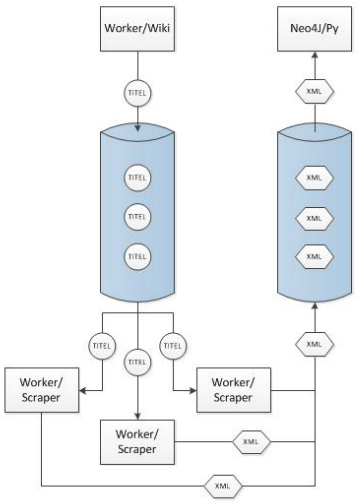

Abbildung 2: Einsatz von Queues im Projekt

2.2.4 EC2-Instanzen

Als Server für dieses Projekt werden die von Amazon angebotenen EC2-Instanzen verwendet. Dabei handelt es sich um Server, die in einer Cloud als Infrastructure as a Service (IaaS) angeboten werden. Amazon stellt insgesamt 11 verschiedene Instanztypen mit unterschiedlicher Hardware und Kosten zur Verfügung. Als Distribution wird das Amazon Machine Image (AMI) Ubuntu Server 12.04.1 LTS x64 verwendet.

Es werden mindestens 3 verschiedene Instanzen für die Umsetzung des Projekts benötigt:

- EC2 High-Memory Extra Large Instanz für die Neo4j-Datenbank
- EC2 Extra Large Instanz für Redis
- Worker-Instanz (Instanztyp abhängig von der Anzahl und Geschwindigkeit der ausgeführten Skripte)

Bei der Wahl einer EC2-Instanz ist es wichtig auf die Sicherheitsgruppe zu achten. Diese verwaltet die freigegebenen Ports des Servers und ist somit für die Kommunikation der Server untereinander von entscheidender Bedeutung.

2.2.5 Frontend

Für das Front-End wurde HTML5, JavaScript und verschiedene Javascript-Bibliotheken genutzt. Die Technologien werden im Folgenden genauer erläutert. Der neue HTML-Standard wird vor allem beim Laden von dynamischen Inhalten und für Animationen genutzt. Insbesondere wird durch HTML 5 der Einsatz von SVG vereinfacht.

Dabei besteht das Grundgerüst der Hauptseite aus relativ wenig HTML-Code. Über CSS wird die konkrete Darstellung der einzelnen Elemente festgelegt. Vorrangig werden Elemente erstellt, die zusammen mit verschiedenen Parametern als Container für weiteren Inhalt dienen.

JavaScript wird für das dynamische Laden von Inhalten, sowie für das Verarbeiten und Darstellen der geladenen Daten genutzt. JQuery, eine JavaScript-Bibliothek, welche die Manipulation der HMTL-Elemente erleichtert, ermöglicht die einfache Erstellung neuer HTML-Elemente und stellt eine einfache Syntax für Ajax-Aufrufe bereit.

2.2.5.1 Dracula

Dracula[6] dient zum Rendern von Graphen mit Hilfe von SVG. Dabei übernimmt die Bibliothek auch das komplexe Anordnen der einzelnen Knoten und der Verbindungen für eine möglichst klare Darstellung. Außerdem ist es möglich, die Knoten zu bewegen und einzelne Knoten mit eigenen Zei-

[6] http://www.graphdracula.net/

chenmethoden zu verknüpfen. Im Projekt werden diese Funktionen genutzt, um die Pfadsuche zu visualisieren. Das Überschreiben der Zeichenmethode wird bei Start- und Endknoten genutzt, um diese hervorzuheben.

2.2.5.2 Chart Draw Tools

Die API von Google[7] stellt verschiedene Diagramme, Graphen oder Tabellen über SVG dar. Tabellen werden allerdings über die HTML-Tabellen-Elemente generiert anstatt über SVG-Elemente. In diesem Projekt wird das Framework verwendet, um die Tabellen der Top 20 darzustellen. Grund dafür ist, dass es leichter zu generieren ist und die Option besteht, weitere Visualisierungen der Daten über die API zu erstellen.

2.2.5.3 HTML Abstract Markup Language (Haml)

Haml ist eine vereinfachte Auszeichnungssprache zur Beschreibung eines XHTML-Dokuments, welche ohne Inline-Programmcode auskommt. Es wird für die Bereitstellung der Statistiken in Verbindung mit JavaScript und CSS verwendet.

2.2.5.4 Neovigator

Bei Neovigator[8] handelt es sich um ein Open-Source-Projekt, bei dem Neography[9] und Processing.js[10] verknüpft werden, um einen Neo4j-Graph anzuzeigen und in ihm dynamisch zu navigieren. Es mussten kleinere Änderungen vorgenommen werden, um Daten der Wikipedia-Artikel anzeigen zu können.

2.2.5.5 Sinatra

Sinatra[11], eine freie Web-Applikations-Bibliothek für Ruby, wird benutzt, um der Internetseite die Daten von Neo4j über eine REST-Schnittstelle zur Verfügung zu stellen. Es wird für jede der Statistiken ein RESTful-Webservice bereitgestellt, welcher Daten als JSON beziehungsweise HTML zurückgibt.

2.3 Realisierung des Prototyps

Die Liste aller Wikipedia-Artikel wird durch ein Bash-Skript heruntergeladen. Die ca. 14MB große Datei wird anschließend entpackt. Danach wird ein Ruby-Skript gestartet, welches die Datei ausliest und die Titel an Redis übergibt.

---

[7] https://developers.google.com/chart/
[8] https://github.com/maxdemarzi/neovigator
[9] leichtgewichtiger Ruby Wrapper für die Neo4j-Rest-API, https://github.com/maxdemarzi/neography
[10] Datenvisualisierung mittels JavaScript, http://processingjs.org/
[11] http://www.sinatrarb.com/

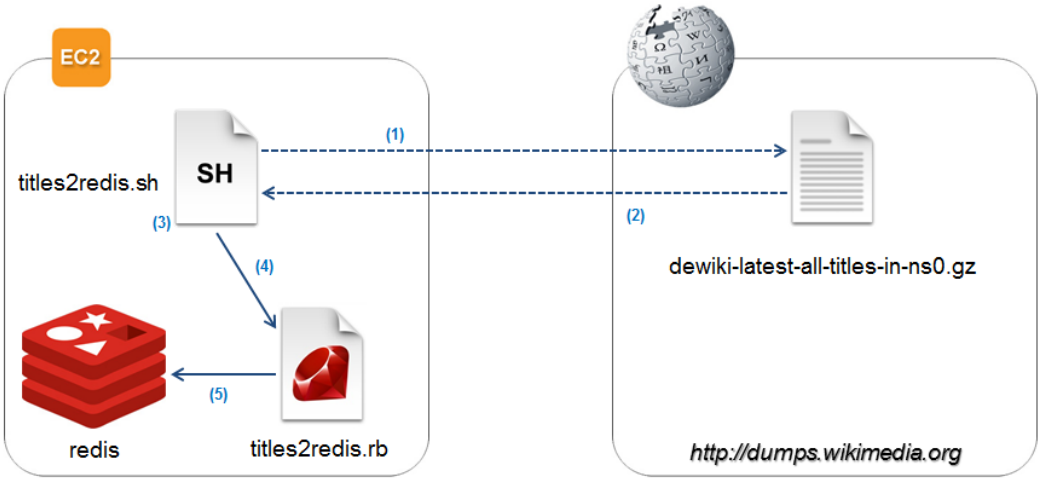

Abbildung 3: Einlesen der Titelliste in Redis

Diese Schritte können sowohl lokal auf der Redis-Instanz als auch von externen Instanzen durchgeführt werden. Dieser Prozess dauert je nach Internetgeschwindigkeit (Download der Datei) ungefähr eine Minute.

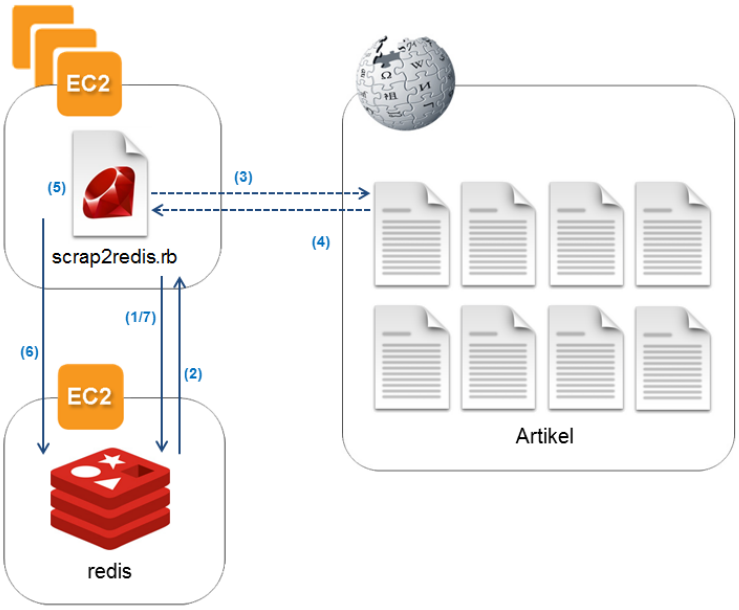

Abbildung 4: Scrapen der Wikipedia-Artikel

Sobald erste Artikel-Titel in Redis gespeichert werden, kann das Ruby-Skript zum Scrapen der Wikipedia-Artikel starten. Dies geschieht in Kombination mit der Bibliothek Nokogiri[12]. Dabei handelt es sich um einen Parser für HTML-Seiten, welcher das Durchsuchen von Dokumenten via XPath oder CSS3-Selektoren erlaubt.

[12] http://nokogiri.org/

Die Wikipedia-Seite wird gescrapt, dessen Daten (Titel, URL, Verlinkungen) in eine XML-Struktur umgewandelt und in einer weiteren Redisliste gespeichert. Diese Daten können je nach Anzahl der Links auf einer Artikel-Seite unterschiedliche Größen annehmen. Das Skript zum Scrapen der Wikipedia-Artikel benötigt im Durchschnitt ungefähr eine Sekunde je Artikel. Dieser Prozess (siehe Abbildung 4) kann jedoch problemlos parallelisiert oder horizontal verteilt werden.

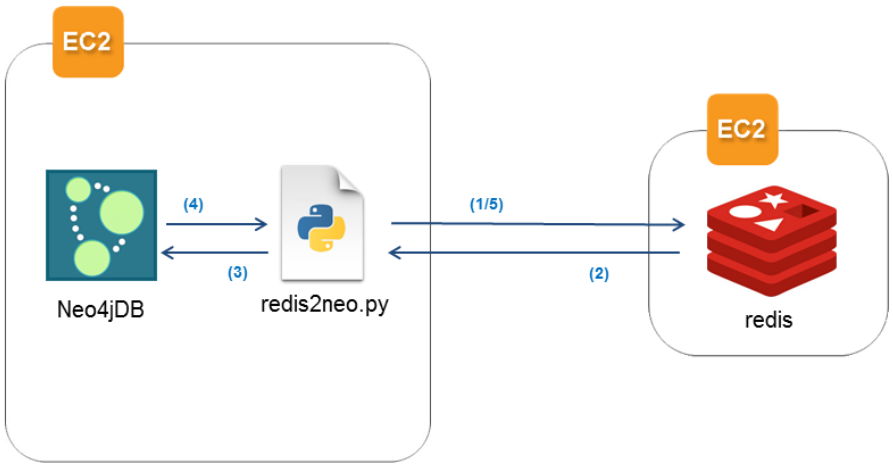

Abbildung 5: Persistente Speicherung in Neo4j

Ein Python-Skript ist für die Speicherung der Wikipedia-Artikel zuständig. Hierfür werden die in Redis gespeicherten XML-Strukturen ausgelesen und über die Embedded-Schnittstelle in der Datenbank gespeichert (siehe Abbildung 5).

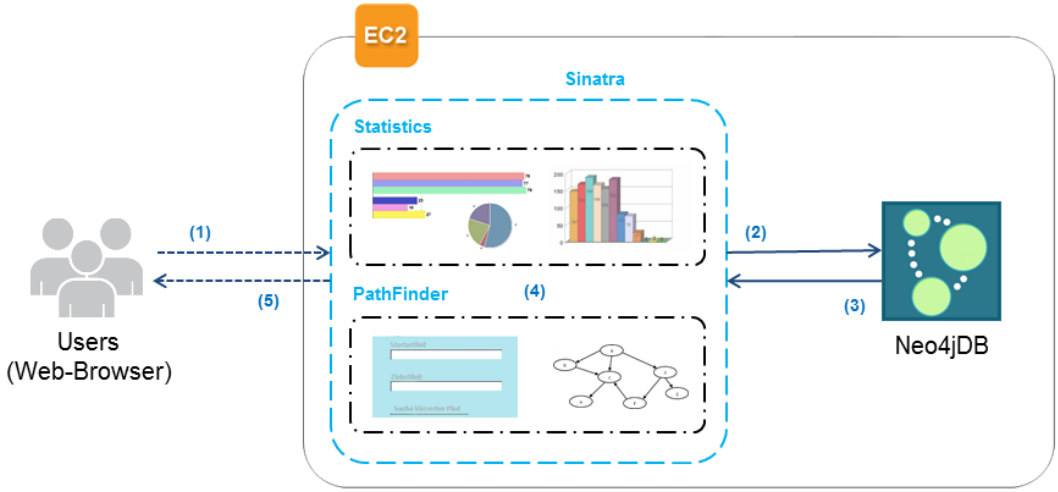

Abbildung 6: Visualisierung der Daten

Zur Visualisierung der persistierten Daten dient eine Webseite. Hierfür werden die Daten vom Neo4j-Server über einen REST-Webservice zur Verfügung gestellt (siehe Abbildung 6).

2.4 Nächste Schritte

Für eine höhere Verfügbarkeit wäre es sinnvoll sowohl Redis als auch Neo4j zu replizieren. Dafür bietet sich das Master-Slave-Prinzip an. Dabei werden alle Aktionen, die auf dem Master-Server durchgeführt werden, auch auf dem Slave-Server ausgeführt. Neo4j bietet zwar eine Möglichkeit der horizontalen Skalierung (Verteilung auf mehrere Instanzen), jedoch wirkt sich dies nur auf Lesezugriffe aus und nicht auf Schreibzugriffe.

Das Skript zum Scrapen der Wikipedia-Artikel sichert fehlgeschlagene Einträge in einer separaten Redis-Liste. Oftmals handelt es sich hierbei um Artikel die nicht mehr existieren, welche jedoch in der Artikelliste noch auftauchen (siehe Liste aller Wikipedia-Artikel Variante 3). Bisher werden die fehlgeschlagenen Einträge nicht automatisiert behandelt. Dies könnte ein weiteres Programm (oder Instanz) als eine Art Kontrollskript übernehmen.

Für das Projekt sind zwei Komponenten extrem wichtig. Sollten Redis oder Neo4j aus welchen Gründen auch immer abstürzen oder nicht zur Verfügung stehen, so verzögert sich der gesamte Ablauf oder lässt sich nicht mehr durchführen und muss per Hand neu adjustiert werden. Für solche Situationen sollten Sicherungsszenarien überlegt und umgesetzt werden. Hierfür könnte beim Ausfall von Redis eine Slave-Instanz angesprochen werden, damit kein Leerlauf entsteht.

Die Verteilung und Anlegung von (Worker-)Instanzen geschieht bisher manuell. Dies könnte durch ein Skript automatisiert werden. Gerade bei extremer horizontaler Skalierung (sehr viele Instanzen) wäre dies eine enorme Zeitersparnis.

Falls die Performance nicht zufriedenstellend sein sollte, so müssen Bottlenecks beseitigt oder andere Technologien evaluiert werden. So könnte beispielsweise eine Embedded-Java-Applikation schneller als Embedded-Python-Applikation sein.

3 Auswertung / Bewertung

3.1 Skalierbarkeit

Es gibt zwei Arten von Skalierbarkeit: vertikal und horizontal. Beide Arten eignen sich für dieses Projekt, jedoch an unterschiedlichen Stellen. Für eine vertikale Skalierung (Verbesserung der Hardware)

können bessere EC2-Instanzen zur Verfügung gestellt werden. Die horizontale Skalierung (Erhöhung der Anzahl der Hardware/Instanzen) eignet sich nicht in jedem Fall, lässt sich schwieriger umsetzen oder bringt in manchen Fällen keine Verbesserung.

Redis und Neo4j profitieren beide durch vertikale Skalierung. In der Theorie sollte Neo4j einen deutlichen Geschwindigkeitszuwachs bei der Verwendung einer SSD erhalten. In der Praxis konnten jedoch lediglich lesende Zugriffe beschleunigt werden. Über die Embedded-Python Anwendung wurde keine Geschwindigkeitsverbesserung festgestellt, obwohl eine deutlich stärkere EC2-Instanz verwendet wurde.

Redis hingegen benötigt einen möglichst großen Hauptspeicher (RAM), da alle Daten in diesem gehalten werden. Für beide Komponenten (Redis und Neo4j) eignet sich keine horizontale Skalierung. Redis läuft lediglich in einem Single-Thread und bietet keine Möglichkeit der Verteilung auf mehrere Threads oder Instanzen. Neo4j könnte nach einem Master-Slave-Prinzip horizontal skaliert werden. Jedoch werden sämtliche schreibende Aktionen auf allen Partitionen durchgeführt. Lediglich eine Performance-Steigerung für lesende Zugriffe wäre gegeben.

Für die horizontale Skalierung eignen sich unsere Worker-Instanzen, also die Skripte zum Scrapen der Wikipedia-Artikel. Ein Skript benötigt für einen Artikel ungefähr eine Sekunde. Es können mehrere Skripte (in verschiedenen Threads) auf derselben Instanz gestartet werden. Je nach Instanz können dies unterschiedlich viele sein. Sollte eine Instanz nicht ausreichende Kapazitäten besitzen oder einen Geschwindigkeitsverlust verzeichnen, so können andere Instanzen helfen diese Last zu verteilen. Je mehr Skripte gleichzeitig laufen, desto mehr Ergebnisse werden in Redis gespeichert und können in die Datenbank überführt werden.

3.2 Bottlenecks

Bottlenecks sind oftmals schwer zu lokalisierende Engpässe in einem Projekt. Je mehr Technologien zum Einsatz kommen, desto mehr Abhängigkeiten entstehen, die zu Problemen führen können. Die Neo4j-Datenbank wurde als größtes Bottleneck identifiziert. Zuerst stellte sich heraus, dass die RESTful-Schnittstelle zu langsam für die riesige Datenmenge von Wikipedia ist. Deshalb wurde dieser Ansatz verworfen und eine Embedded-Applikation entwickelt. Diese arbeitete zwar deutlich schneller als gegenüber REST, aber trotzdem noch zu langsam. Selbst die beste EC2-Instanz brachte hierbei keinen Performance-Zuwachs gegenüber einer Extra Large Instanz. Dies führt zu der Schlussfolge, dass das geschriebene Skript in Python uneffektiv implementiert wurde. Eventuell werden zu viele einzelne oder unnötige Transaktionen versendet, die den Prozess verlangsamen. Leider bietet Neo4j

keine Möglichkeit die Datenbank horizontal zu skalieren, sodass die Schreiblast auf mehrere Server verteilt werden könnte.

Redis konnte als Bottleneck ausgeschlossen werden. Ein Benchmark-Test zeigte, dass ungefähr 5000 schreibende und lesende Zugriffe je Sekunde bei Daten mit 20.000 Bytes durchgeführt werden können (siehe Tabelle 1). Dieser Durchsatz wird jedoch bei weitem nicht ausgeschöpft.

Tabelle 1: redis-benchmark -h 176.34.212.62 -n 100000 -d 20000 -q

Aktion	Requests per second
PING_INLINE	17876.30
PING_BULK	17992.08
SET:	4712.54
GET	4936.32
INCR	18001.80
LPUSH	4743.83
LPOP	4936.81
SADD	17975.91
SPOP	17866.71
LPUSH	4711.65
LRANGE_100 (first 300 elements):	42.87
LRANGE_300(first 450 elements):	14.31
LRANGE_500(first 600 elements):	9.54
LRANGE_600	7.15
MSET (10 keys)	431.38

3.3 Probleme

Jede Technologie bringt eigene Tücken und Schwierigkeiten mit sich. Insbesondere Performance-Probleme waren eine Schwierigkeit dieses Projekts. Daten konnten über eine Neo4j-RESTful-Schnittstelle nicht schnell genug in die Datenbank gespeichert werden. Dies konnte zwar durch eine Embedded-Applikation optimiert werden, jedoch wird der gleichzeitige Aufruf von RESTful-Anfragen auf die Datenbank, die gerade mit der Embedded-Applikation verbunden ist, abgelehnt. Doppelte Instanziierungen auf einen Datenbestand sind erlaubt.

Auf einer Large-Instanz ist bei mehreren Testdurchläufen der Speicher vollgelaufen und der OOM-Killer hat den Redis-Server-Prozess terminiert. Dabei wurde der Datenbank-Dump beschädigt und konnte nicht mehr vollständig geladen bzw. mit dem Server synchronisiert werden. Die komplette EC2-Instanz musste neu gestartet werden, da Redis die Verbindung dauerhaft abwies (Connection refused).

Dieses Problem wurde mit einer X2.Large Instanz (High-Memory) gelöst, jedoch stellten sich hier Kommunikationsprobleme mit Redis ein (Timeout), die die Skripte auf den Worker-Instanzen regelmäßig unterbrachen. Allerdings scheint dies ein Problem unter Unix zu sein und daher nicht Redis spezifisch[13].

Eine weitere Schwierigkeit stellten die fehlenden Rechte in der AWS-Management-Konsole dar. Es konnten keine Sicherheitsgruppen geändert oder angelegt werden. Daher konnten beispielsweise nur die standardmäßig freigegeben Ports verwendet werden. Glücklicherweise gab es eine vordefinierte Sicherheitsgruppe für Redis.

Darüber hinaus stellt Amazon nur eine unzureichende Anzahl an unterschiedlichen EC2-Instanzen zur Verfügung. Vielmehr wäre eine freie Konfiguration von Komponenten wünschenswert.

4 Ergebnis / Fazit

Das Projekt Scrapedia konnte erfolgreich umgesetzt werden. Es existiert ein valider Datenbestand vom kompletten deutschen Wikipedia in einer Neo4j-Graphendatenbank. Darüber hinaus konnten erste Statistiken wie zum Beispiel meist und nie verlinkte Artikel ermittelt werden. Als Visualisierung können einzelne Artikelseiten und deren Verknüpfungen mit Hilfe des Neovigators angezeigt werden. Zusätzlich gibt es die Möglichkeit die kürzesten Pfade zwischen zwei Knoten aufzulisten und visualisiert darzustellen.

Es wurde eine funktionierende Architektur einer verteilten Anwendung zum Scrapen von Webseiten entwickelt und implementiert. Dabei wurden verschiedene Technologien wie Redis und Neo4j eingesetzt. Es können auf dem aktuellen Datenbestand Pfade von einem Artikel zu einem anderen performant gesucht und visualisiert dargestellt werden. Allgemeine Statistiken und eine Visualisierung der Verzweigungen aller Knoten können ebenfalls über die entwickelte Webseite aufgerufen und dargestellt werden.

Wie in jedem Projekt gab es auch in diesem Probleme. Insbesondere die Performanz der Datenbank erzeugt ein bisher ungelöstes Zeitproblem. Dennoch könnten die erzeugten Projektergebnisse für weitere Arbeiten, wie zum Beispiel die Entwicklung eines Wikipedia-Spiels, verwendet werden.

[13] http://stackoverflow.com/questions/11847900/how-to-deal-with-read-timeout-in-redis-client

Literaturverzeichnis

Hoff, Todd. 2009. Neo4j - A Graph Database That Kicks Buttox. *http://highscalability.com.* [Online] 06 13, 2009. [Cited: 01 06, 2013.] http://highscalability.com/neo4j-graph-database-kicks-buttox.

Kumar, Ravi and Raghavan, Prabhakar and Rajagopalan, Sridhar and Sivakumar, D. and Tompkins, Andrew and Upfal, Eli. 2000. The Web as a graph. *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems.* s.l. : ACM, 2000.

Montag, David. 2012. Hardware Sizing for Neo4j. *neo4j.org.* [Online] 05 2012. [Cited: 01 06, 2013.] http://watch.neo4j.org/video/41111155.

Redis Documentation - Replication. [Online] [Cited: 01 08, 2013.] http://redis.io/topics/replication.

Redis, Ubuntuusers -. http://wiki.ubuntuusers.de/Redis. [Online] [Cited: 01 08, 2013.]

Webber, Jim. 2011. On Sharding Graph Databases. *Jim Webber's Blog.* [Online] 02 16, 2011. [Cited: 01 07, 2013.] http://jim.webber.name/2011/02/on-sharding-graph-databases/.

—. 2011. Scaling Neo4j with Cache Sharding and Neo4j HA. *Jim Webber's Blog.* [Online] 02 23, 2011. [Cited: 01 07, 2013.] http://jim.webber.name/2011/02/scaling-neo4j-with-cache-sharding-and-neo4j-ha/.