Ostfalia
Hochschule für angewandte Wissenschaften

Masterarbeit

Entwicklung einer Steuerungskomponente zur Priorisierung von Aufträgen für verteilte Webcrawls

Studiengang:
Medieninformatik Master (Online)

vorgelegt von

Jens Gäbeler
Matrikelnr.: 70453756

am 19. Juli 2020
an der Ostfalia Hochschule

Erstprüfer: Prof. Dr.-Ing. Nils Jensen

Zweitprüfer: Prof. Dr. rer. nat. habil. Torsten Sander

suma-ev

osf open search foundation

Zusammenfassung & Abstract

Zusammenfassung

Das Ziel der vorliegenden Masterarbeit ist die Erstellung einer zentralisierten Steuerungskomponente, die innerhalb eines verteilten Web-Crawling-Systems agiert und unterschiedliche Strategien zur Priorisierung und Partitionierung von URLs und Domains einsetzen kann.

Mithilfe einer systematischen Literaturanalyse werden existierende Strategien gesammelt und kategorisiert. In einer prototypischen Umsetzung werden die Grundfunktionalitäten der Steuerungskomponente und eine Auswahl der ermittelten Strategien implementiert. Ausgehend von der Steuerungskomponente wird ein Testsystem mit den relevanten Web-Crawling-System-Komponenten URL-Datenbank, verteilte Fetcher-Simulatoren und Ergebnissammler erstellt. Eine Teststeuerung erprobt die Ausführung der Strategien im Testsystem. Die Erprobung erfolgt auf den in der Arbeit erarbeiteten Metriken *Anteil besuchter Seiten, Aktualität* und *Auslastung.*

Das Ergebnis der Arbeit ist ein Prototyp, der für die Produktion weiterentwickelt werden kann und ein Testsystem, das mit zusätzlichen Strategien erweitert werden und mit bestehenden Strategien verglichen werden kann.

Schlüsselbegriffe: Verteiltes Web Crawling, Priorisierung, Partitionierung, Simulation

Abstract

Target of this master thesis is the construction of a centralized scheduler, which operates within a distributed web crawling system and utilizes various strategies to prioritize and partition URLs and domains.

A systematic literature review is used to gather and categorize existing strategies. A prototypic scheduler with basic functionality and the implementation of selected strategies will be developed. Starting from the scheduler a testing system with relevant web crawling components (frontier database, distributed fetcher simulators and resource collector) is developed. An additional Test Executor runs and compares the strategies inside the testing system. The comparison will be done using the developed metrics *ratio of visited pages*, *freshness* and *utilization*.

The result of this thesis is a working prototype, which can be further developed for production and a testing system, which can be enhanced and compared with additional and existing strategies.

Keywords: distributed web crawling, prioritization, partitioning, simulation

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Glossar

Abdeckung (engl. Coverage)

Verhältnis besuchter Seiten gegenüber allen gewünschten Seiten im Internet.

Aktualität (engl. Freshness)

Verhältnis aktueller Seiten in der Datenbank gegenüber allen Seiten in der Datenbank.

Crawler

Programm zum automatisierten Durchsuchen und Verarbeiten von Webseiteninformationen im Internet.

Fetcher

Programm zum automatisierten Durchsuchen von Webseiten im Internet.

Frontier

Datenspeicher für URLs innerhalb eines Web-Crawling-Systems.

Fully Qualified Domain Name (abgekürzt FQDN)

Die FQDN beschreibt den vollständigen Domainname in der Form `www.example.com` (The Internet Society 1987).

Invertierter Web Index

Ein Index, in dem für vorhandene Begriffe jeweils das vorkommende Dokument genannt ist. Ähnlich dem Stichwortverzeichnis eines Buches.

Scheduler

Steuerungsprogramm zur Regelung der zeitlichen Ausführung von Aufgaben.

Seed

Initiale URL-Liste innerhalb des Frontiers oder als separate Komponente.

Universally Unique Identifier (abgekürzt UUID)

Die UUID beschreibt eine Zeichenkette der Länge 128-bit, die zur globalen Identifizierung von Objekten in der Softwareentwicklung benutzt werden kann (The Internet Society 2005).

1 Einleitung

1.1 Zweck

Mithilfe des Open Web Index (OWI) möchte die *Open Search Foundation (OSF)* eine europäische Alternative zu den Webindizes der Firmen *Google*, *Microsoft*, *Baidu* und *Yandex* schaffen. Dieser neue und offene Index soll sowohl von öffentlich-rechtlichen als auch von kommerziellen Organisationen genutzt werden können, um vielfältige Dienste erstellen und anbieten zu können. Im Folgenden soll dazu ein Beitrag im Teilbereich Web Crawling geleistet werden.

Schätzungen zufolge beinhaltet der Index von Google 60 Milliarden und der Index von Bing 13 Milliarden Einzelseiten (Kunder 2019). Einen Index in ähnlicher Größenordnung zu erstellen, setzt voraus, diverse Herausforderungen untersucht und Lösungen dafür bereitgestellt zu haben. Eine dieser Herausforderungen ist, dass die Abarbeitungsliste der zu verarbeitenden URLs sehr groß wird und weder eine vollständige Abdeckung aller existierenden Seiten im Index repräsentiert werden kann, noch alle bekannten Seiten in angemessener Aktualität erneut erfasst werden können.

Die geplante modulare Bauweise des Open Web Index ermöglicht die Konzentration auf einzelne Aspekte, wie zum Beispiel den Teilbereich der Steuerung von Crawl-Aufträgen. Dieser Teilbereich ist bisher nicht in der Systemarchitektur des Open Web Indexes integriert. Die im Architekturkonzept des OWI bestehenden Komponenten *OWI Crawler* und *OWI Web Index* (siehe Abbildung 1) stehen in Verbindung mit der geplanten Steuerungskomponente.

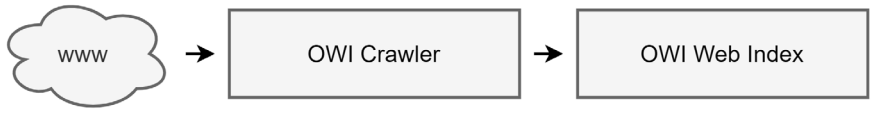

Abbildung 1: Teilbereich des Open Web Index (Auszug nach Huss et al. 2017, S. 5)

Die Arbeit trägt dazu bei, einen Überblick über das Thema Web Crawling in den Bereichen der kontinuierlich laufenden und inhaltlich nicht beschränkten Crawler zu schaffen. Zudem wird durch den Entwurf und den Prototyp ein detaillierter Einblick in den Entwicklungsprozess eines Web Crawlers gegeben. Der Prototyp soll mit Abschluss der Masterarbeit der Open Search Foundation als erweiterbare Komponente übergeben werden.

1.2 Ziel der Arbeit

Mit der Masterthesis soll eine Steuerungskomponente konzipiert und erstellt werden. Diese Steuerungskomponente soll in einem verteilten Web-Crawling-System agieren und mithilfe einer REST-Schnittstelle von Web Crawlern angeforderte URL-Listen bereitstellen. Das Ziel der Masterthesis ist es, die *Abdeckung*, die *Aktualität* und den *Durchsatz* des Gesamtsystems jeweils einzeln und in Kombination zu maximieren, ohne dass fremde Ressourcen überbeansprucht werden. Geschehen soll dies durch die Beeinflussung der Priorisierung und der Partitionierung der Gesamtliste.

1.3 Vorgehensweise

Um das Ziel zu erreichen, wurden die folgenden fünf Arbeitspakete erstellt.

AP1 Analyse zum Stand der Forschung: Es sollen die internen und externen Einflussfaktoren, die die Abdeckung und Aktualität und den Durchsatz eines Web-Crawling-Systems beeinflussen, gefunden und zusammengefasst werden. Zugehörige Strategien sollen ebenfalls gegenübergestellt werden. Des Weiteren sind Metriken und Evaluationsmethoden zu finden, die einen Vergleich der Strategien ermöglichen.

AP2 Technische Vorbereitung: Vorbereitend für die prototypische Realisierung wird in diesem Arbeitspaket bereits ein minimal funktionierendes Programm erstellt. Dabei wird keine Logik oder Strategie implementiert, sondern lediglich die Infrastruktur.

AP3 Entwurf: In Verbindung der Erkenntnisse aus AP1 und AP2 soll in diesem Arbeitspaket ein Entwurf für die Steuerungskomponente entstehen. Der Entwurf endet in einer Reihenfolge zu implementierender Strategien und wird von einem Experten auf Machbarkeit überprüft.

AP4 Realisierung: Das vierte Arbeitspaket beinhaltet die Implementierung nach der Reihenfolge der im Entwurf erstellten Vorgaben.

AP5 Erprobung: Es folgt die Ausführung und Auswertung verschiedener Strategien im entwickelten Prototyp.

1.4 Aufbau der Arbeit

Die Arbeit besteht aus acht Kapiteln. In *Kapitel 2* wird die Aufgabenstellung im Detail beschrieben. Im darauf folgenden *Kapitel 3* werden die Grundlagen von Web Crawlern, soweit dies für die folgenden Kapitel relevant ist, erläutert. Die Literatur zu aktuellen Strategien wird in *Kapitel 4* analysiert. Darauf aufbauend entsteht im *Kapitel 5* aus den gesammelten Anforderungen ein Entwurf für die Steuerungskomponente, deren Realisierung in *Kapitel 6* dokumentiert wird. Es folgt *Kapitel 7*, in dem die Methode des Vergleichs der implementierten Strategien erklärt wird und die Ergebnisse der Vergleiche gegenübergestellt werden. Die Arbeit schließt in *Kapitel 8* mit dem Fazit und einem Ausblick ab.

2 Aufgabenstellung

Das Kapitel Aufgabenstellung beschreibt die zu bearbeitenden Arbeitspakete im Detail und gibt eine Abgrenzung. Die einzelnen Arbeitspakete verfolgen übergreifend das Ziel der Vereinbarung von hoher Abdeckung, Aktualität und Durchsatz in einem verteilten Web-Crawling-System. Dafür wird zu Beginn eine Analyse bestehender Systeme vorgenommen und die technische Infrastruktur für die Realisierung vorbereitet. In einem weiteren Schritt werden die Ergebnisse in einem Entwurf für eine Steuerungskomponente zusammengefasst. Der Entwurf stellt die Grundlage für die anschließende prototypische Entwicklung dar. Abschließend werden durch Tests mit dem Prototyp und einem Testsystem Vergleichswerte unterschiedlicher Strategien erzeugt und gegenübergestellt.

2.1 Analyse (AP1)

Im ersten Arbeitspaket werden die relevanten wissenschaftlichen Grundlagen zum Thema Web Crawling zusammengefasst und der Stand der Forschung bezüglich Priorisierungs- und Verteilungstechniken von URLs erläutert. Als Grundlagen werden die Aufgaben, Anforderungen und Herausforderungen an Web Crawler und der Steuerungskomponente aufgenommen. Ebenfalls erfolgt eine Analyse bestehender Web-Crawling-System-Architekturen, die für die geplante Umsetzung geeignet sind. Im weiteren Verlauf der Analyse werden die beeinflussenden Faktoren für die Abarbeitung von URL-Listen beschrieben und Strategien mit einer positiven Beeinflussung auf die Metriken Abdeckung, Aktualität und Durchsatz zusammengefasst.

Als Ergebnis der Analysephase entsteht eine Übersicht an Priorisierungs- und Verteilungsstrategien, die als Grundlage für den Entwurf (AP3) der Steuerungskomponente verwendet werden.

Für die Analyse der wissenschaftlichen Literatur wird die Systematische Literaturanalyse (engl. Systematic Literature Review) nach Kitchenham (2007) angewendet. Das dazugehörige Protokoll kann in *Anhang A* eingesehen werden. Für die Analyse wurden vier Forschungsfragen entwickelt und als Suchabfrage in vier Online-Datenbanken für wissenschaftliche Publikationen eingegeben. Die Ergebnisse wurden gesammelt, analysiert, priorisiert und gefiltert. Zusätzliche Anwendung der *Forward-* und *Backwardsnowballing*-Technik ergab eine finale Literaturliste mit 48 Ergebnissen.

2.2 Technische Vorbereitung (AP2)

Im zweiten Arbeitspaket wird eine minimale Version der Steuerungskomponente, mit der per REST-Schnittstelle kommuniziert werden kann, realisiert. Durch die Erstellung und Distribution werden die technische Infrastruktur, die Programmiersprache, Frameworks und Bibliotheken sowie die zu verwendenden Werkzeuge überprüft und ausgewählt.

2.3 Entwurf (AP3)

Die Ergebnisse der Analysephase und der technischen Vorbereitung werden in diesem Arbeitspaket zu einem Softwareentwurf zusammengefasst.

Es entsteht ein Architekturentwurf, der sich in den modularen Aufbau der OWI-Dienste eingliedert, ein Datenmodell- und ein API-Entwurf. Die gegliederten Strategien werden entsprechend ihrer Umsetzungskomplexität priorisiert und in eine Implementierungsreihenfolge gebracht.

Nach einer Machbarkeitsprüfung der Entwürfe durch einen Experten der *Open Search Foundation* werden Änderungswünsche angepasst und die Entwürfe für das nächste Arbeitspaket freigegeben.

2.4 Realisierung (AP4)

Die Realisierung des Prototyps erfolgt auf Basis der geprüften Entwürfe (AP3) und der Auswahl der technischen Infrastruktur und Werkzeuge (AP2).

Die einzelnen Strategien werden nach der vorbestimmten Implementierungsreihenfolge mit agilen Entwicklungsmethoden umgesetzt. Dadurch wird gewährleistet, dass jederzeit ein lauffähiges System zur Verfügung steht. Für die Programmierung wird ein testgetriebener Entwicklungsansatz[1] angewandt.

2.5 Erprobung (AP5)

Der finalisierte Prototyp wird ausgeführt und die Ergebnisse verschiedenen Strategien einander gegenübergestellt. Durch die Gegenüberstellung sollen die Funktionalitäten des Prototyps validiert werden und eine optimale Strategie oder eine optimale Kombination mehrerer Strategien gefunden werden.

2.6 Abgrenzung

Die Steuerungskomponente soll in zentralisierter Form erstellt werden. Der Ansatz, dass die Steuerungskomponente selbst verteilt ist, wird nicht verfolgt. Diese Eingrenzung hat folgende Vorteile:

1) Es gibt keinen Kommunikationsbedarf zwischen den Web Crawlern und somit auch nicht die Anforderung, einen Kanal mit allen anderen Web Crawlern schaffen zu müssen.
2) Durch die fehlende Kommunikation zwischen den Web Crawlern können alle auftretenden Verbindungen und Datenflüsse von der Steuerungskomponente protokolliert werden. Dies erleichtert die Vergleichbarkeit der verschiedenen Strategien.
3) Die Web Crawler sind unabhängig voneinander. Die Gesamtanzahl ist für die einzelne Instanz nicht relevant.

Die Herausforderungen dieser zentral laufenden Komponente sind allerdings der hohe Speicher-, Rechen- und Netzwerkzugriffsbedarf. Des Weiteren wird die Entscheidung für einen zentralen Ansatz oder für einen verteilten Ansatz der Speicherung der URL-Daten nicht getroffen. Für die Arbeit wird davon ausgegangen, dass die Datenspeicherung als gegeben gesehen werden kann. Die URL-Liste soll durch SQL oder SQL-ähnliche Syntax abgefragt werden können.

[1] Test Driven Development nach Beck (2015).

3 Grundlagen Web Crawler

Das Kapitel behandelt die grundlegenden Informationen zum Thema. In einem ersten Überblick wird der Forschungsbereich Web Crawling vorgestellt, im Weiteren die zu berücksichtigenden Aspekte bei der Konzeption von Web Crawlern. Abschließend werden unterschiedliche Architekturen und Typen von Web Crawlern vorgestellt.

3.1 Übersicht

Das Unterkapitel führt in den Forschungsbereich Web Crawling ein. Dazu wird dieser Bereich in den ihn umgebenden Forschungsbereich eingeordnet. Im Weiteren werden der Prozessablauf und die Aufgaben von Web Crawlern beschrieben.

3.1.1 Einordnung

Informationsgewinnung (engl. Information Retrieval, kurz IR) befasst sich mit dem Abrufen und der Verarbeitung von Informationen aus verschiedensten Quellen. Aufgeteilt wird die Informationsgewinnung in traditionelle und automatisierte IR (siehe Abbildung 2). Informationsgewinnung erfolgt traditionell durch algebraische oder statistische Modelle. In der automatisierten Form wird IR verwendet, um eine sehr hohe Anzahl an Informationen verarbeiten zu können.

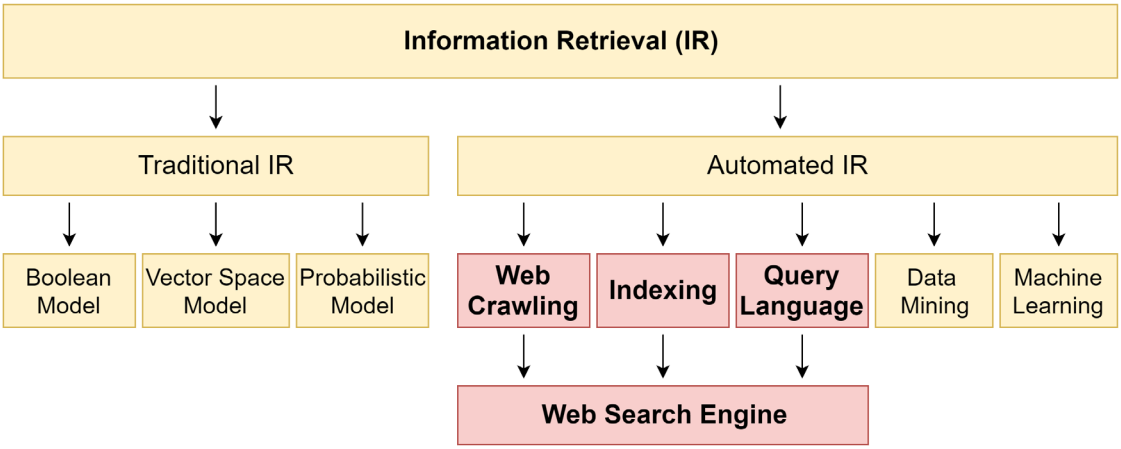

Abbildung 2: Kategorisierung von Lösungsansätzen für die Informationsgewinnung (in Anlehnung an Saini und Arora 2016, S. 2635 und Cambazoglu und Baeza-Yates 2015, S. 6)

Für eine Suchmaschine sind die in Abbildung 2 hervorgehobenen Bereiche *Web Crawling*, *Indexing* und *Query Language* zwingend nötig. Der Zusammenhang zwischen den Bereichen in einer Suchmaschine ist in Abbildung 3 dargestellt. Das *Web-Crawling*-System bezieht Informationen aus dem Internet und sendet diese an einen Datenspeicher, das *Web Repository*. Das *Indexing System* greift auf das *Web Repository* zu und speichert ausgewählte Informationen in einem weiteren Datenspeicher,dem *Web Index* ab. Das *Query Processing System* greift auf den *Web Index* zu, um Anfragen von *Benutzern* mit geeigneten *Ergebnissen* beantworten zu können (Cambazoglu und Baeza-Yates 2015, S. 6).

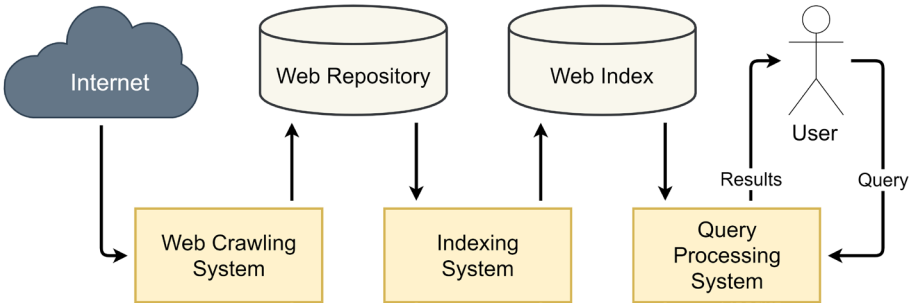

Abbildung 3: Web Crawler als eine der drei Komponenten einer Web-Suchmaschine (in Anlehnung an Cambazoglu und Baeza-Yates 2015, S. 6)

Für die Wissenschaft ist die Erforschung und Verbesserung von Web Crawlern ein sehr interessanter Bereich. Mechanismen für parallellaufende und verteilte Systeme benötigen geeignete und umfassende Algorithmen und Datenstrukturen. Diese gilt es zu erforschen, insbesondere, da die Technik von kommerziellen Crawlern aus Wettbewerbsgründen nicht frei zugänglich ist (Ueda 2013, S. 66).

3.1.2 Ablauf

Der Prozess eines einfachen Web-Crawling-Systems läuft dabei in der folgenden wiederkehrenden Schleife ab (Olston und Najork 2010, S. 178):

1) Eine URL aus einer URL-Liste wird aufgerufen[2]
2) Die aufgerufene Seite wird heruntergeladen
3) Die heruntergeladene Datei wird nach neuen URLs durchsucht
4) Alle gefundenen URLs werden in der URL-Liste gespeichert

Zusätzlich sind in einfachen Web-Crawling-Systemen bereits die folgenden Komponenten enthalten und interagieren untereinander (Lee et al. 2008, S. 3):

- Die *URLSeen-Liste:* Eine Liste an bereits besuchten URLs, um diese nicht wiederholt abzurufen.
- Der *Robots.txt-Cache:* Ein Cache mit Robots.txt-Daten, um die Zugriffe auf gewünschte und unerwünschte Bereiche des Website-Besitzers berücksichtigen zu können.
- Der *DNS-Cache:* Ein Cache mit der Auflösung der Domains zu IP-Adressen, um wiederholte Abfragen zu externen DNS Providern zu verringern.

3.1.3 Metriken

Web Crawler besitzen zwei grundlegende Aufgaben. Einerseits sollen unbekannte URLs gefunden und verfolgt werden, andererseits sollen bereits bekannte URLs nochmals auf Änderungen untersucht werden. Dies dient im ersten Fall der Vergrößerung der Abdeckung (engl. Coverage) und im zweiten Fall der Erhöhung der Aktualität (engl. Freshness) der Seiten (Cambazoglu und Baeza-Yates 2015, S.

[2] Damit das Web-Crawling-System starten kann, muss mindestens eine URL bereitgestellt sein.

10). Als Abdeckung wird dabei der Anteil der Seiten von allen erwünschten Seiten definiert, die erfolgreich abgerufen werden konnten. Aktualität wird definiert als der Anteil der Seiten im Web Repository, deren Inhalte den Seiten im aktuellen Internet gleichen (Olston und Najork 2010, S. 194).

Die Abdeckung und die Aktualität werden als Gleichungen wie folgt dargestellt:

Gleichung 1: Abdeckung der Seiten einer Datenbank

$$Abdeckung = \frac{Anzahl\ besuchter\ Seiten\ in\ der\ URL\text{-}Datenbank}{Anzahl\ gewünschter\ Seiten\ im\ Internet}$$

Gleichung 2: Aktualität der Seiten in einer Datenbank

$$Aktualität\ = \frac{Anzahl\ aktueller\ Seiten\ in\ der\ URL\text{-}Datenbank}{Anzahl\ aller\ Seiten\ in\ der\ URL\text{-}Datenbank}$$

Diesen beiden Metriken stehen die beiden Faktoren Wachstum und Änderungsrate des Internets gegenüber, welche die Abdeckung und die Aktualität der gesammelten Daten negativ beeinflussen (Cambazoglu et al. 2008, S. 32). Die Größe selbst, das Wachstum und die Änderungsrate des Internets tragen dazu bei, das eine vollständige Abdeckung ebenso wenig möglich ist wie eine vollständige Aktualität. Es wird eine Kompromisslösung benötigt, um beide Aufgaben zufriedenstellend bearbeiten zu können. Das Problem ähnelt der Balancefindung zwischen Erkunden (engl. exploration) von potentiell relevanten Seiten und Auswerten (engl. exploitation) von bereits bekannten Seiten (Olston und Najork 2010, S. 178).

Als allgemeines Maß, welches durch Veränderung sowohl die Abdeckung als auch die Aktualität beeinflussen kann, steht der Durchsatz (engl. throughput) zur Verfügung. Der Durchsatz ist definiert als die Menge der Information, die in einem bestimmten Zeitraum übertragen werden kann (Cambazoglu et al. 2008, S. 33).

Als Gleichung wird der Durchsatz dargestellt als:

Gleichung 3: Durchsatz einer Komponente in einem Netzwerk

$$Durchsatz\ = \frac{Anzahl\ übertragener\ Daten}{Zeiteinheit}$$

Den Metriken von Web-Crawling-Systemen stehen die Metriken von Webseiten gegenüber. Mit diesen besteht die Möglichkeit eine Webseite durch einen Wert auszudrücken und somit vergleichbar und sortierbar zu machen. Für die Sortierung von URL-Listen in Web-Crawling-Systemen werden Kombinationen der drei Metriken *Wichtigkeit*, *Relevanz* und *Änderungsrate* verwendet (Olston und Najork 2010, S. 202). Die *Wichtigkeit* (engl. importance) einer Seite wird relativ zu den anderen Webseiten berechnet, dies geschieht häufig durch Graph-Algorithmen wie beispielsweise dem PageRank (Brin und Page 1998). Die *Relevanz* drückt aus, inwieweit eine Seite der gesetzten Aufgabe eines Web Crawlers entspricht (Cambazoglu et al. 2008, S. 32). Die Metrik der *Änderungsrate* beschreibt in welchem Zeitraum sich die Inhalte einer Seite verändert haben (Cambazoglu et al. 2008, S. 32).

Tabelle 1: Metriken für Web Crawler und Webseiten

Metriken für Web Crawler		Metriken für Webseiten	
Abdeckung	engl. coverage	Wichtigkeit	engl. importance
Aktualität	engl. freshness	Relevanz	engl. relevance
Durchsatz	engl. throughput	Änderungsrate	engl. dynamicity

Tabelle 1 fasst die Metriken zusammen. Eine Auswahl zu berücksichtigender Metriken geschieht zu Beginn der Analyse in Kapitel 4.

3.2 Anforderungen

Im Folgenden werden Anforderungen an Web Crawler, die sich für einen realen Einsatz eignen, beschrieben. In erster Linie wird hierfür die Skalierbarkeit gefordert, aber auch ein respektvoller Umgang mit fremden Ressourcen ist wichtig.

3.2.1 Skalierbarkeit

Web Crawler müssen skalierbar sein, um mit den äußerst umfangreichen Datenmengen des Internets effizient umgehen zu können. Durch Hinzufügen von weiteren Web-Crawler-Instanzen soll es möglich sein, einen höheren Durchsatz für das Herunterladen der Seiten zu realisieren. Um dies bewerkstelligen zu können, ist es nötig, Web Crawler verteilt zu betreiben und die Last allen verwendeten Knoten in geeigneter Weise zuzuteilen. Einzelne Knoten sollen dabei geringe Ressourcen verbrauchen. Zudem soll eine Priorisierung von hochwertigen Seiten eine Abarbeitung ermöglichen, die nicht vollständig sein muss (Ueda 2013, S. 67).

3.2.2 Schonung fremder Ressourcen

Die in der englischen Literatur als *politeness* bezeichnete Anforderung erwartet von Web Crawlern, andere Nutzer des Internets nicht zu beeinträchtigen. Andere Nutzer sind einerseits die vom Crawler besuchten Web Server, aber auch die Infrastruktur des Internets selbst und indirekt die Endanwender. Eine Überbeanspruchung von fremden Ressourcen kann zur Folge haben, dass die Bereitstellung von Inhalten an Endanwender verlangsamt wird oder aber auch, zum Schutz der Endanwender, dass der Web Crawler vom Zugriff auf einen Web Server blockiert wird.

Um fremde Ressourcen nicht zu überlasten, sollten Fehlermeldungen in geeigneter Form berücksichtigt werden und URLs zu fehlerhaften Seiten von weiteren Abrufen ausgeschlossen werden. Zudem sollten die Intervalle der Abrufe zu gleichen Servern lang genug sein und gleiche Seiten nicht zu oft wiederholt heruntergeladen werden. Dies gilt ebenso für die Auflösung der Domain-Namen bei DNS-Diensten, die durch eine Zwischenspeicherung in einem Cache reduziert werden kann. Für Webseiten existiert die freiwillig von den Betreibern der Seite zur Verfügung stellbare `robots.txt`-Datei. Mit dieser können die gewünschten Intervalle vom Web Crawler erfasst und berücksichtigt werden (Ueda 2013, S. 67). In dieser Datei kann durch Setzen des Crawl-Delay-Wertes ein Wert in Sekunden angegeben werden, den der Betreiber der Webseite den Web Crawlern als Intervall zwischen zwei Abrufen vorschlägt. Nach Statistiken aus dem Jahr 2007 besitzen 28 % aller Webseiten eine valide `robots.txt`-Datei, 20 % der validen Dateien besitzen einen Wert für das `crawl-delay`.

Die gewünschte Zeitfrist zum nächsten Abruf beträgt darin durchschnittlich 7 Sekunden (Kolay et al. 2008, S. 1171 f.).

Die Anforderung verschärft sich, wenn statt eines einzelnen Web Crawlers sehr viele verteilte Web Crawler einem Index zuarbeiten. Wenn keinerlei Kontrolle über die abzurufenden URLs jedes Crawlers durchgeführt wird, kann es passieren, dass ein Großteil der Crawler gleichzeitig auf einen Web Server zugreift. Es besteht somit die Gefahr eines wie in Abbildung 4 dargestellten *Distributed Denial of Service (DDoS)*, einem Angriff der dem von Botnetzen ähnelt.

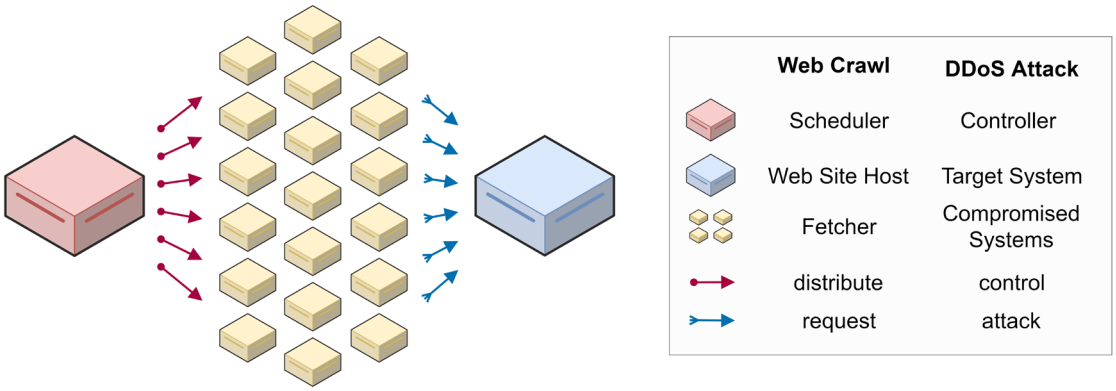

Abbildung 4: Architektur eines verteilten Web Crawls bzw. eines DDoS Angriffs

3.2.3 Weitere Anforderungen

In der Literatur werden weitere Anforderungen an Web-Crawling-Systeme genannt. Unter diesen sind die Verwendung eines geeigneten Algorithmus für die Abschätzung der Zeitfrist zwischen dem Abruf einer bestimmten Seite, eine einfache Konfigurierbarkeit und eine einfache Anpassbarkeit (Ueda 2013, S. 66). Die unterschiedlichen Anforderungen werden in der folgenden Tabelle 2 nach Nennung sortiert zusammengefasst.

Tabelle 2: Anforderungen an Web Crawler aus (Ueda 2013, S. 66 f.)

Nr.	Anforderung
A1	Skalierbarkeit
A2	Schonung fremder Ressourcen
A3	Abschätzung der Änderungsrate
A4	Einfache Konfigurierbarkeit
A5	Einfache Anpassbarkeit

3.3 Herausforderungen und Lösungsansätze

3.3.1 Herausforderungen

Zu den expliziten Anforderungen an Web-Crawling-Systeme, kommen weitere Herausforderungen hinzu, die sich durch die Interaktion mit den externen Systemen im Internet ergeben. Diese reichen von der Verlängerung der Crawler-Intervalle über böswillige Seitenbetreiber bis hin zu überoptimierter User Experience.

Sind die Intervalle zwischen den Webseitenabrufen desselben Servers zu groß gewählt kann der Web Crawler unter Umständen nicht seine volle *Auslastung* erreichen (Cambazoglu und Baeza-Yates 2015, S. 12). Ähnliche Auswirkungen haben *Verzögerungsangriffe* (engl. Delay Attacks). Bei diesen bösartigen Angriffen wird serverseitig ein künstlicher Zeitraum vor den Antworten erzeugt. Dadurch sind die Crawler für eine kurze Zeit nicht beschäftigt und der durchschnittliche Durchsatz eines Web Crawlers wird verringert (Cambazoglu und Baeza-Yates 2015, S. 22).

Eine weitere Herausforderung sind Webseiteninhalte, die dynamisch generiert werden. Seiten mit dynamischen Inhalten können als bösartig und gutartig kategorisiert werden. Beide Fälle können als *Crawler Traps* bezeichnet werden. Bösartige Crawler Traps bestehen, wenn eine Webseite dynamisch unzählige Unterseiten generiert, die untereinander verlinkt sind. Die Betreiber versuchen durch dieses Vorgehen, ein Thema im Index prominenter zu besetzen. Durch Verlinkung von allen dynamisch generierten Unterseiten auf eine Hauptseite soll andererseits die Berechnung von Wichtigkeit-Scores, wie beispielsweise dem PageRank, positiv beeinflusst werden. Diese Form von *Crawler Traps* nennt sich *Link Farm* (Olston und Najork 2010, S. 226). Die Auswirkungen können gravierend sein: Durch die zufallsgesteuerte Generierung von 1.000 Links zu Domains mit Wildcard-DNS Eintrag können bei der Anwendung der Breitensuche eines Crawlers innerhalb von drei Ebenen eine Milliarde fehlerhafte Einträge in die URL-Datenbank gelangen (Cui et al. 2018, S. 3).

Ebenfalls problematisch für Web Crawler sind aber auch die gutartigen *Crawler Traps*, wie beispielsweise Online-Kalender mit Verlinkungen zwischen den Monaten, wodurch ebenfalls kein Crawl-Ende gefunden werden kann (Olston und Najork 2010, S. 226 f.). Es stellt sich somit die Herausforderung der Auswahl der Inhalte. Wie kann qualitativ hochwertiger Inhalt von weniger qualitativen, nicht relevanten, redundanten oder bösartigen Inhalten unterschieden werden (Olston und Najork 2010, S. 178 f.)? Dabei liegt das Problem hauptsächlich bei großen Spam-Seiten. Kleine Seiten mit Spaminhalten beeinflussen die Leistung von Web Crawlern in einem zu vernachlässigenden Maß (Lee et al. 2008, S. 7).

Weitere externe Faktoren, die für Web Crawler und deren Entwickler herausfordernd sind, betreffen die Inhalte von Webseiten, die heruntergeladen werden sollen. Dies ist beispielsweise der Umgang mit sogenannten *Soft 404*-Fehlern, welche von Webseiten bei fehlenden Seiten zurückgegeben werden. Im Gegensatz zum Standard, erfolgt hierbei nicht die Rückgabe des Rückgabecodes `404 Not Found`, sondern eine angepasste Fehlerseite zusammen mit dem Rückgabecode `200 OK`. Ein Web Crawler erkennt diese Seite somit als Erfolg, wird die Seite unnötigerweise im Index ablegen und auch wieder besuchen. Ebenfalls die Web-Crawler-Leistung und den realen Durchsatz verringernd ist die Spiegelung (engl. Mirroring) von Webseiten und somit der vielfache Abruf von gleichen Inhalten (Cambazoglu und Baeza-Yates 2015, S. 23).

Eine weitere Herausforderung ist die Existenz von Seiten mit vielen Millionen Unterseiten, wie z.B. Facebook (1,4 Mrd. Seiten) oder eBay (57 Mio. Seiten).[3] Würden diese sequentiell abgerufen, würden bei Abrufintervallen von 5 Sekunden für die Unterseiten von Facebook 222 Jahre und für die Unterseiten von eBay 9 Jahre benötigt. Bei einer parallelen Bearbeitung von 1.000 Crawlern und

[3] Die Anzahlen wurden mithilfe der Suchabfragen „site:facebook.com“ bzw. „site:ebay.com“ auf https://www.google.de ermittelt. Es wurde die Anzahl der Ergebnisse verwendet. Abgerufen am 27.12.2019.

einer Reduzierung des Intervalls auf 1 Sekunde würde ebay.com innerhalb von 16 Stunden vollständig übertragen werden können. Der Download der Seiten von Facebook benötigt mit dieser Einstellung dennoch 16 Tage, erst die Erhöhung der Anzahl der Crawler auf 25.000 ermöglicht eine vollständige Übertragung innerhalb von 16 Stunden.

3.3.2 Lösungsansätze

Ein Lösungsansatz, der für alle Herausforderungen eine Verbesserung verspricht, ist die Erhöhung des Durchsatzes durch mehr Leistung, zum Beispiel durch mehr parallele Web Crawler oder eine erhöhte Datenrate.

Bekannte oder unbekannte Verzögerungen während der Übertragung können somit ausgeglichen werden. Einer *geringen Auslastung* kann durch die Anwendung von mehreren parallelen Prozessen in einem Web Crawler entgegengewirkt werden. Dadurch wird allerdings die Größe des RAM in erhöhtem Maße beansprucht (Lee et al. 2008, S. 2). Auswirkungen von *Verzögerungsangriffen* können durch eine allgemeine Herunterstufung aller Seitendownloads mit langen Wartezeiten verringert werden. Möglich ist die Herunterstufung ab einem gesetzten Wert für die Wartezeit. Webseiten mit höheren Wartezeiten werden in der Folge weniger oft besucht (Cambazoglu und Baeza-Yates 2015, S. 22).

Crawler Traps & *Link Farmen* sind aufgrund ihrer ähnlichen Struktur von Webseiten wie Facebook, Twitter oder eBay nicht unterscheidbar (Cambazoglu und Baeza-Yates 2015, S. 22).

Spiegelungen von Webseiten können erkannt werden, indem die Linkstruktur aller Webseiten analysiert und die Ähnlichkeit untereinander gemessen wird (Cambazoglu und Baeza-Yates 2015, S. 23). Ein Vergleich jeder Webseite mit jeder anderen ist jedoch sehr aufwendig. Eine Reduzierung der Vergleiche auf Seiten mit ähnlichen Inhalten kann hierbei die Performanz erhöhen.

Die Erkennung von *Soft 404 Fehlermeldungen* ist dagegen einfacher zu realisieren: Zu jeder Domain werden mehrere unübliche URLs generiert und aufgerufen. Dadurch kann das Muster dieser Fehlerseiten erlernt und gespeichert werden (Cambazoglu und Baeza-Yates 2015, S. 23).

Die folgende Tabelle listet die Herausforderungen und Lösungsansätze nach Auftreten im Text sortiert auf:

Tabelle 3: Herausforderungen und Lösungsansätze von Web-Crawling-Systemen

Nr.	Herausforderung	Lösungsansatz
H1	Erreichung hoher Auslastung	Durchsatz erhöhen durch parallele Prozesse & Herunterstufung von Seiten mit langen Wartezeiten
H2	Erkennung von Crawler Traps & Link Farmen	Kein Ansatz
H3	Erkennung von Webseiten Spiegelungen	Linkstrukturanalyse und Ähnlichkeitsmessung
H4	Umgang mit Soft-404 Fehlern	Muster von Fehlerseiten erkennen
H5	Abruf von Webseiten mit vielen Unterseiten	Durchsatz oder Abrufintervalle erhöhen

3.4 Architekturen

Web Crawler können grundlegend in drei aufeinander aufbauende Architekturmodelle eingeordnet werden. Diese sind in der Reihenfolge von einfach bis komplex: sequenzielle, parallele und verteilte Web Crawler. Des Weiteren können Web Crawler anhand ihrer Ziele unterschieden werden.

3.4.1 Sequenzielle Web Crawler

Die Vorgehensweise eines sequenziellen Web Crawlers wurde bereits in der Web-Crawler-Übersicht dargestellt. Etwas genauer wird hier der Prozess der Entdeckung von Webseiten ergänzt. In diesem Prozessablauf können Webseiten in drei Status unterteilt werden. Es wird unterschieden in *heruntergeladene* (engl. downloaded), *entdeckte* (engl. discovered) und *unentdeckte* (engl. undiscovered) Seiten.

Die folgende Abbildung 5 stellt die Unterschiede dieser drei Status im Crawling-Prozess dar. *Heruntergeladene* Seiten sind im Web Repository verfügbar und wurden auf neue URLs untersucht. Diese dabei *entdeckten* URLs werden im sogenannten Frontier abgespeichert. Die restlichen URLs, die weder im Web Repository noch im Frontier gespeichert sind, besitzen den Status *unentdeckt* (Cambazoglu und Baeza-Yates 2015, S. 14).

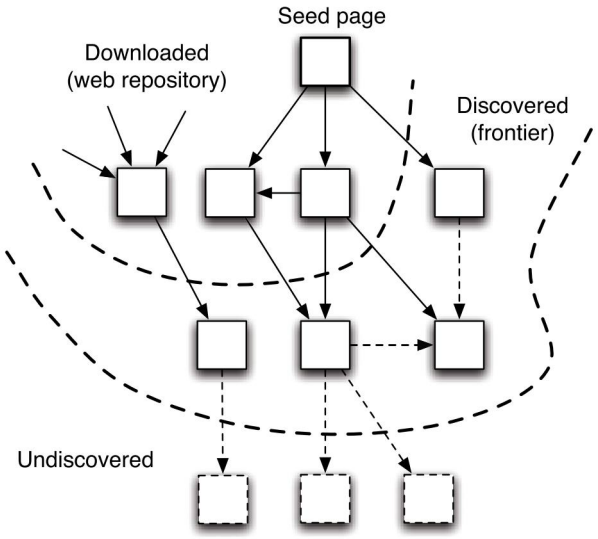

Abbildung 5: Seitenstatus im Crawling-Prozess (Cambazoglu und Baeza-Yates 2015, S. 15)

Trotz einiger gelöster Herausforderungen ist Web Crawling ein ressourcenintensiver Prozess. Insbesondere die Prozessorleistung, der Festplattenspeicher, Arbeitsspeicher und das Netzwerk werden in hohem Maße beansprucht (Cambazoglu et al. 2008, S. 31).

3.4.2 Parallele Web Crawler

Bei einem parallelen Web Crawler werden die Prozesse aufgeteilt, so dass mehrere Web Crawler gleichzeitig agieren können. Durch die Verteilung der URL Abrufe reduziert sich die Beanspruchung der Ressourcen des Prozessors und des Arbeitsspeichers (Cambazoglu et al. 2008, S. 31). Die Engpässe liegen somit bei der Netzwerkverbindung und dem Festplattenspeicher.

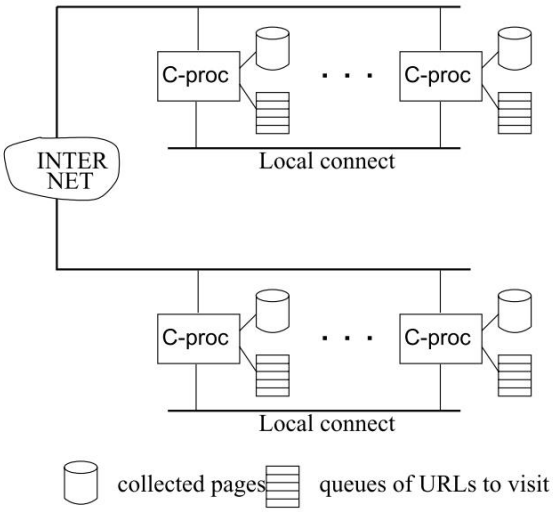

Abbildung 6: Parallelisierung in Web Crawlern (Cho und Garcia-Molina 2002, S. 126)

Die Parallelisierung kann lokal oder im Internet verteilt stattfinden. In Abbildung 6 sind diese beiden Formen zu erkennen. Die lokale Parallelisierung erfolgt durch mehrere Instanzen auf demselben Server. Alternativ erfolgt eine zusätzliche Parallelisierung durch Verteilung der Crawler auf mehrere Internetknoten (Cho und Garcia-Molina 2002, S. 126). Diese zweite Art ist Teil des folgenden Unterkapitels 3.4.3 Verteilte Web Crawler.

Bei der Parallelisierung entstehen die folgenden drei Herausforderungen zu den bereits für sequenzielle Web Crawler bestehenden. Erstens kann eine *Überlappung* (engl. Overlap) entstehen, falls mehrere parallel arbeitende Web Crawler dieselbe Webseite herunterladen und somit Ressourcen ohne Zusatznutzen verbrauchen. Zweitens kann es passieren, dass die Qualität der Webseiten in der lokalen Frontier-Liste nicht optimal berechnet wurde, da lediglich eine *Teilmenge aller Informationen* zur Verfügung steht. Durch die dritte Herausforderung – dem *Kommunikationsbedarf* – können die Probleme der ersten beiden Herausforderungen reduziert bzw. eliminiert werden. Allerdings steigt

der Bedarf der Kommunikation (engl. Communication Bandwidth) relativ zur Menge der verwendeten Web Crawler (Cho und Garcia-Molina 2002, S. 124). Je nach Anwendungsfall werden hier unterschiedliche Lösungsansätze gewählt.

Die Koordination der Web Crawler kann auf drei verschiedene Wege erfolgen. Im einfachsten Fall erfolgt *keine Koordination*. Alle Web Crawler agieren dann unabhängig voneinander. Dadurch wird keine Kommunikation benötigt, allerdings kommt es hierbei zu *Überlappungen*. Alternativ kann eine *dynamische Zuordnung* angewandt werden. Diese erfordert einen zentralen Koordinator und einen Partitionsalgorithmus, welcher die Seiten oder Domains einzelnen Instanzen zuordnet. Der Nachteil dieser Methode ist, dass der Koordinator selbst zum Engpass des Web-Crawling-Systems werden kann, wenn viele Web-Crawler-Knoten mit Partitionen versorgt werden müssen. Eine weitere Alternative ist die *statische Zuordnung*, in der das Internet für alle Instanzen vorpartitioniert vorliegt. In dieser Variante ist kein zentraler Koordinator nötig, da die Web-Crawler-Knoten sich gegenseitig die URLs der zugewiesenen Partition zusenden (Cho und Garcia-Molina 2002, S. 126 f.).

3.4.3 Verteilte Web Crawler

In einem verteilten Web Crawler erfolgt eine Skalierung durch das Hinzufügen von neuen Knoten an geographisch unterschiedlichen Orten. Jeder Knoten erhält somit eine eigene Netzwerkanbindung. Es bleibt, im Gegensatz zum lokal parallellaufenden Web Crawler, lediglich der Engpass der Datenspeicherung für das Web Repository (Cambazoglu et al. 2008, S. 31).

Durch die höhere kumulierte Netzwerkgeschwindigkeit eines verteilten Web Crawlers erhöht sich der Durchsatz, wodurch sowohl Abdeckung als auch Aktualität des Web Repositories verbessert werden können (Cambazoglu und Baeza-Yates 2015, S. 20). Die geographische Verteilung der Web-Crawler-Instanzen ermöglicht es außerdem, genau die Seiten zu crawlen, die sich in geographischer Nähe befinden (Quoc et al. 2015, S. 389). Dadurch reduzieren sich die Downloadzeiten einzelner Abrufe.

Der Synchronisationsaufwand des verteilten Web-Crawling-Systems steigt, da sich die Kommunikationszeiten zwischen den Knotenpunkten im Gegensatz zu zentralen parallellaufenden Web Crawlern verlängern. Dementsprechend ist der Aufwand für ein Systemdesign, welches die Reduzierung der Kommunikationskosten als eine der primären Anforderungen besitzt, hoch (Quoc et al. 2015, S. 389).

Kommunikationskosten können eingespart werden, indem die Übertragung der besuchten Webseiten an den zentralen Index optimiert durchgeführt wird. Dies kann entweder durch Sammlung und Komprimierung der Übertragung geschehen oder dadurch, dass nur die Unterschiede zur letzten Version gesendet werden. Zu einer hohen Reduzierung würde außerdem die Übertragung lediglich der Auswertungen anstatt der kompletten Seite führen (Cho und Garcia-Molina 2002, S. 127 f.).

Zudem muss ein geeignetes Vorgehen für die Partitionierung bzw. Neupartitionierung der zu besuchenden URLs verwendet werden. Bei einem hinzugefügten oder entfernten Knoten im Web-Crawling-System sollten möglichst wenig Änderungen bei den URL-Zuweisungen einzelner Web Crawler vorkommen (Cambazoglu et al. 2008, S. 34).

3.4.4 Web-Crawler-Unterteilung nach Zielen

Web Crawler werden alternativ zur Architektur in ihren Zielen unterschieden. Eine Unterscheidung geschieht bezüglich des Inhalts oder des Ablaufes. Abbildung 7 zeigt diese Taxonomie, wobei ein Web Crawler gleichzeitig mehreren Unterpunkten zugeordnet sein kann.

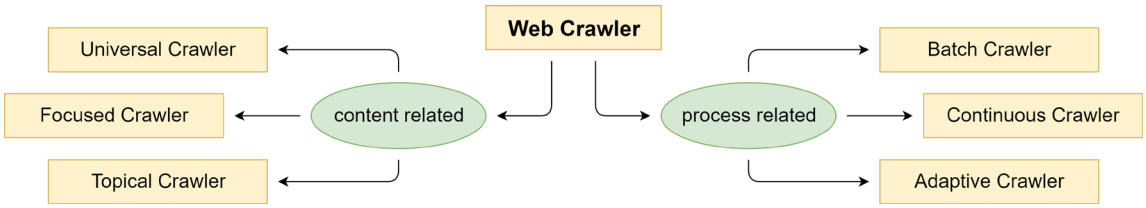

Abbildung 7: Web-Crawler-Arten (in Anlehnung an Kumar et al. 2017, S. 3, Olston und Najork 2010, S. 212 f.)

Universelle, fokussierte und thematische Web Crawler sind inhaltsbezogen. Universelle Crawler besitzen keine Restriktionen auf spezielle Webseiten oder Themen. Im Gegensatz dazu durchsuchen fokussierte und thematische (engl. topical) Crawler das Internet nach vorgegebenen Anfragen oder Themen und ignorieren andere Seiten.

Eine andere Einordung erfolgt über den Ablauf des Web Crawlers. Batch-Crawler durchlaufen eine vorgegebene Liste an URLs und werden manuell gestartet. *Kontinuierlich laufende Crawler* dagegen aktualisieren ihre Abarbeitungsliste während des Betriebs und laufen bis sie manuell abgeschaltet werden (Kumar et al. 2017, S. 3). Im Vordergrund steht hierbei das Ziel, eine gute Balance zwischen Abdeckung und Aktualität zu leisten. Es werden sowohl Erstbesuche von unbekannten Seiten als auch erneute Besuche von alten Seiten durchgeführt (Olston und Najork 2010, S. 215). Ein adaptiver Crawler verwendet aktuelle Informationen zur Bewegung im Internet und trifft Entscheidungen kurzfristig (Olston und Najork 2010, S. 212 f.).

4 Systematische Literaturanalyse

Im folgenden Kapitel erfolgt die Analyse der wissenschaftlichen Literatur. Es werden Strategien und Techniken betrachtet, die die Abdeckung, die Aktualität und den Durchsatz von Web-Crawling-Systemen erhöhen. Dazu werden zu Beginn der Rahmen und die Inhalte dieser Analyse beschrieben. Die Analyse betrifft thematisch die beiden Bereiche: *Priorisierung* der Frontier-Liste und *Verteilung* der URLs an die Fetcher. Das Kapitel schließt mit einer Übersicht der Ergebnisse ab.

4.1 Rahmen der Analyse

Für die Sammlung und initiale Auswertung von wissenschaftlichen Publikationen zum Thema, wurde eine systematische Literaturanalyse durchgeführt, deren Protokoll in *Anhang A* nachgelesen werden kann. Der Rahmen der Analyse orientiert sich an den Vorgaben für die zu entwickelnde Komponente und an der Auswahl der zu berücksichtigenden Metriken, Anforderungen und Herausforderungen.

4.1.1 Systemvorgaben

Das zu entwickelnde System wird die Steuerungskomponente in einem universellen, verteilten und kontinuierlich laufenden Web Crawler sein. Aufgrund der Vielzahl an Strategien wird für die folgenden Kapitel eine Auswahl für die *Priorisierung* und *Verteilung* getätigt. Die Auswahl orientiert sich einerseits an der Architektur des später zu entwickelnden Prototyps, ist andererseits offen genug, um Vergleichstests bei der Erprobung zu ermöglichen. Es werden beispielsweise Strategien für Single Server- und Batch-Crawler betrachtet, da diese gegebenenfalls als Strategien für Teilkomponenten verwendet werden können. Das Web-Crawling-System in Gesamtheit, die Fetcher-Komponenten und die Steuerungskomponente besitzen folgende Vorgaben:

Das *Web-Crawling-System* soll

- kontinuierlich & themenunabhängig,
- unabhängig von Nutzerabfragen,
- unabhängig von Suchergebnisrängen sein.

Die *Fetcher-Komponenten* sollen

- geographisch verteilt sein,
- ressourcensparend laufen.

Die *Steuerungskomponente* soll

- zentralisiert laufen,
- Zugriff auf Meta-Informationen von URLs besitzen,
- per REST API ansprechbar sein.

4.1.2 Auswahl der Metriken

Aus den Metriken (siehe Tabelle 1 auf Seite 8) werden lediglich die für die Systemvorgaben passenden Metriken ausgewählt. Dies sind alle Web-Crawler-Metriken (*Abdeckung*, *Aktualität* und *Durchsatz*)

und die Webseiten-Metriken *Wichtigkeit* und *Änderungsrate*. Die Webseiten-Metrik *Relevanz* wird nicht weiter berücksichtigt, da eine Relevanzbedingung für universelle Web Crawler nicht besteht und auch Suchergebnisseiten für Nutzer, die relevante Ergebnisse zu ihrer Suchanfrage erhalten wollen, nicht das Ziel des *Open Web Indexes* sind.

Zudem werden in der Analyse weitere Informationen über Webseiten berücksichtigt, durch deren Verwendung kann eine Erhöhung der Abdeckung, der Aktualität oder des Durchsatzes ermöglicht werden. Im Weiteren werden die Informationen: Anzahl der Unterseiten, vorgegebene *Verzögerungszeit* (Crawl Delay), *IP-Adresse* und der *Hashwert der FQDN* verwendet.

4.1.3 Aufgaben

In den Grundlagen wurden bereits die Anforderungen und Herausforderungen von Web-Crawling-Systemen – im Weiteren zusammenfassend als Aufgaben bezeichnet – aufgenommen. Dies erfolgte in den Kapiteln 3.2 & 3.3. Im aktuellen Unterkapitel werden nun die einzelnen Punkte, die bereits in Tabelle 2 (Seite 9) und Tabelle 3 (Seite 12) aufgelistet wurden, zusammengefasst und den zwei Kategorien Priorisierung und Verteilung zugewiesen. Ebenfalls darin enthalten ist, inwieweit die jeweiligen Aufgaben mithilfe einer der ausgewählten Metriken aus dem vorigen Unterkapitel 4.1.2 gemessen werden können. Zudem werden die Punkte bereits anhand der Systemvorgaben (Kapitel 4.1.1) für die Umsetzungsreihenfolge sortiert.

Die folgende Tabelle 4 listet ausgewählte Aufgaben auf, die durch Priorisierungsstrategien bearbeitet werden können.

Tabelle 4: Aufgaben für Priorisierungsstrategien

Priorität	Aufgabe	Referenz	Web-Crawler-Metrik	Webseiten-Metrik
P1	Erreichung hoher Auslastung	H1	Abdeckung	Wichtigkeit
P2	Abschätzung der Änderungsrate	A3	Aktualität	Änderungsrate
P3	Skalierbarkeit	A1	Abdeckung	Wichtigkeit

Die *Erreichung einer hohen Auslastung* des Gesamtsystems ist das wichtigste Ziel (P1). Durch die Messung der Abdeckung kann die Qualität der Erreichung gemessen werden. Eine hohe Auslastung soll möglichst auf wichtige Seiten im Internet beschränkt werden. Eine Berechnung der Wichtigkeit aller Seiten ermöglicht eine dahingehende Sortierung. Das zweitwichtigste Ziel ist die *korrekte Abschätzung der Änderungsrate* von Einzelseiten (P2). Diese ist nötig, um eine hohe Aktualität des Web Repositories mit geringem Aufwand zu erreichen. Eine *Skalierbarkeit* (P3) des Systems zu gewährleisten ist das nächstwichtige Ziel. Die Priorisierung von wichtigen Seiten ermöglicht die Reduzierung von Seiten mit nicht gewünschten Inhalten und somit eine Vergrößerung der Abdeckung von Seiten mit gewünschten Inhalten.

In der nächsten Tabelle 5 sind ausgewählte Aufgaben aufgelistet, die für die Verteilung der URL-Liste relevant sind.

Tabelle 5: Aufgaben für Verteilungsstrategien

Priorität	Aufgabe	Referenz	Web-Crawler-Metrik	Webseiten-Metrik
V1	Schonung fremder Ressourcen	A2	-	Abrufintervall
V2	Erreichung hoher Auslastung	H1	Durchsatz	-
V3	Skalierbarkeit	A1	Durchsatz	-

Das wichtigste Ziel bei der Verteilung ist die Anforderung der *Schonung fremder Ressourcen* (V1). Diese ist nicht durch die aufgenommenen Web-Crawler-Metriken messbar. Auswirkungen auf die Abdeckung sind erst verzögert sichtbar, beispielsweise durch eine Sperrung der IP-Adresse der Fetcher-Komponente. Die Berücksichtigung des im *Robots Exclusion Protocol* der Webseite vorgegebenen Abrufintervalls (engl. Crawl Delay) ist erforderlich. Eine *hohe Auslastung* (V2) kann durch die Verteilung selbst und die Parallelisierung von Abarbeitungslisten einer einzelnen Fetcher-Komponente erreicht werden. Eine Reduzierung der Downloadzeiten mithilfe einer Zuweisung von Webseiten zu „nahen" Fetcher-Komponenten kann zusätzlich bewirken, dass inaktive Zeit reduziert wird. Die Anforderung der *Skalierbarkeit* (V3) kann durch geeignete Verteilungsstrategien erreicht werden, die den Overhead bei Änderungen der Fetcher-Komponentenanzahl minimieren. Mithilfe des kumulierten Durchsatzes aller Crawler können beide Anforderungen gemessen werden.

Durch die Zusammenfassung der Systemvorgaben, die Auswahl der Metriken und die Zuweisung der Aufgaben anhand der Anforderungen und Herausforderungen aus den vorigen Kapiteln können nun die Strategien im Detail betrachtet werden. Die beiden Tabellen werden für die weiteren Abschnitte die Grundlage bilden.

4.2 Priorisierungsstrategien

Im Folgenden wird erörtert, wie URLs in der Frontier-Liste in geeigneter Weise sortiert werden können, so dass sinnvolle Seiten im Crawl-Vorgang vor weniger sinnvollen Seiten verarbeitet werden. Sinnvoll in der Hinsicht, dass eine möglichst hohe Abdeckung und angemessene Aktualität der Internetseiten im Web Repository gegeben wird. Einfluss darauf haben hierbei die Webseiten-Metriken Wichtigkeit und Änderungsrate.

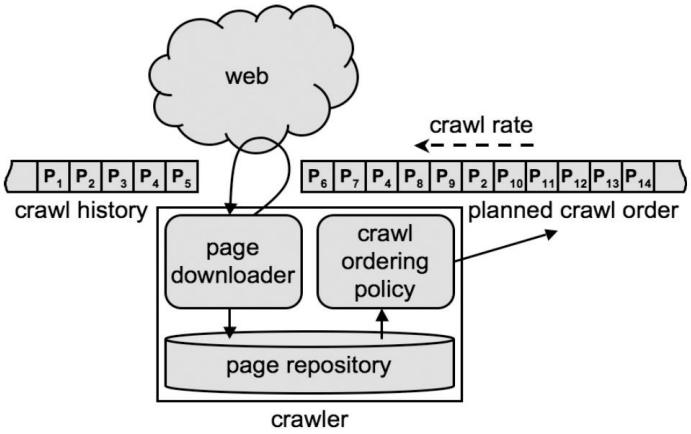

Abbildung 8: Priorisierungsstrategie eines Web Crawlers (Olston und Najork 2010, S. 195)

Im Allgemeinen ist die Priorisierungsstrategie (engl. crawl ordering policy) in einem kontinuierlich ablaufenden Web Crawler wie in Abbildung 8 lokalisiert. Eine anstehende URL wird aufgerufen und die zurückgesendeten Daten in das Web Repository gespeichert. Entsprechend der Priorisierungsstrategie werden URLs aus den Seiten des Web Repositories an einer berechneten Stelle in der Abarbeitungsliste abgelegt (Olston und Najork 2010, S. 195).

Im Weiteren wird auf Priorisierungsstrategien, welche die Metriken *Abdeckung* und *Aktualität* eines Web Repositories optimieren, näher eingegangen.

4.2.1 Strategien zur Maximierung der Abdeckung

Eine hohe Abdeckung ist in erster Linie das Ziel eines Batch-Crawlers. Aber auch ein inkrementeller Web Crawler kann durch Strategien zur Erhöhung der Abdeckungen verbessert werden. Insbesondere in der initialen Phase des Web Crawlers, wenn Seiten zum ersten Mal gecrawlt werden (Olston und Najork 2010, S. 202).

Die häufigsten dafür verwendeten Strategien sind die *Breitensuche* (engl. Breadth First Search), die Priorisierung nach *Anzahl eingehender Links* (engl. Indegree oder backlink-count) und die Priorisierung nach *PageRank* (PR) bzw. PR-Varianten oder PR-Abschätzungen (Olston und Najork 2010, S. 205 f.). Die *Breitensuche* ist ein Graph-Algorithmus bei dem, im Falle eines Web Crawlers, alle Seiten in der Reihenfolge in der sie entdeckt wurden, heruntergeladen werden. Diese Strategie ist sehr einfach umzusetzen und benötigt wenig Ressourcen für die Berechnung. Allerdings ist die *Breitensuche* anfällig für Seiten, die unerwünschte Inhalte enthalten (Cui et al. 2018, S. 2). Komplexer dagegen ist die Umsetzung der *Indegree*-Strategie, bei der die Seiten nach der Anzahl eingehender, bereits bekannter Links in die Abarbeitungsliste einsortiert werden. Eine ähnliche Strategie ist die Priorisierung nach der *PageRank*-Bewertung einer Seite, hier erfolgt eine im Gegensatz zur *Indegree*-Stategie nochmals komplexere Berechnung und ebenso ein höherer Ressourcenverbrauch. Aufgrund dieser Rechenintensivität, bei der alle Knoten in einem Graph in jedem Schritt neu berechnet werden müssen, wurden bereits verschiedene Varianten und Abschätzungsverfahren entwickelt (Cui et al. 2018, S. 2). Eine ressourcensparende Variante ist der *Batch-PageRank.* Hierbei wird der PageRank nicht nach jeder neuen URL, sondern nach größeren Abständen (bspw. nach 100.000 Aktualisierungen) neu berechnet (Baeza-Yates et al. 2005, S. 867). Eine weitere Alternative ist das bevorzugte Abrufen von

Webseiten, die viele Unterseiten besitzen (engl. *larger-sites-first*). Diese Strategie wirkt dem entgegen, dass gegen Ende eines Batch-Crawlvorgangs alle Webseiten mit wenig Unterseiten bereits abgerufen wurden. Somit besteht zu diesem Zeitpunkt keine Möglichkeit mehr, die Leerläufe zwischen den Abrufen von Unterseiten großer Webseiten mit anderen Unterseiten zu befüllen (Baeza-Yates et al. 2005, S. 867).

Abbildung 9 stellt dies in der Gegenüberstellung anhand eines Beispiels dar. Die Anzahl der zusammengehörigen Pakete ist jeweils im Paket eingetragen. Für dieses Beispiel wurde eine Wiederabrufverzögerung von 2,5 Sekunden gewählt. Bei Priorisierung auf „Große-Seiten-Zuerst" entsteht eine Leerlaufzeit von 1,5 Sekunden innerhalb der benötigten 11,5 Sekunden. Bei der „Kleine-Seiten-Zuerst"-Strategie entsteht eine Leerlaufzeit von 4,5 Sekunden und es werden für den Crawlvorgang 14,5 Sekunden benötigt. Die Leerlaufzeit ist um den Faktor drei vergrößert.

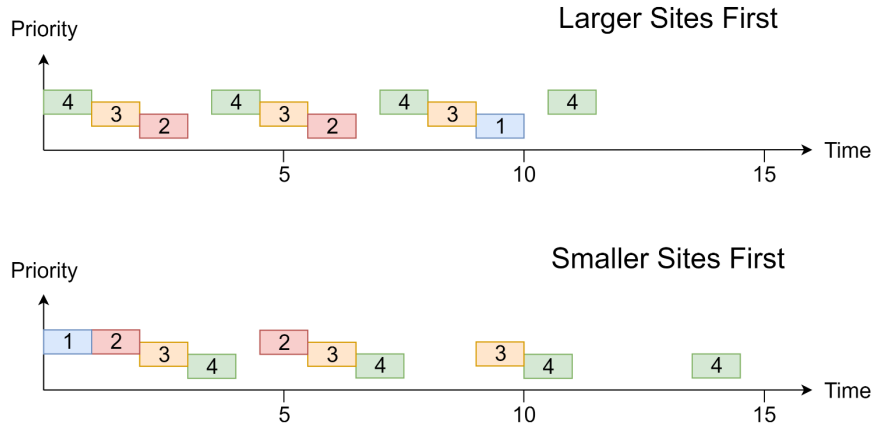

Abbildung 9: Dauer der Strategie Große-Seiten-Zuerst gegenüber der Strategie Kleine-Seiten-Zuerst

Wie sich diese Strategie bei nicht-endenden Web Crawlern verhält, die kontinuierlich neue Seiten erhalten, ist nicht bekannt.

4.2.2 Strategien zur Maximierung der Aktualität

Die Aktualität im Web Repository zu erhöhen kann lediglich durch das kontinuierliche Wiederabrufen von bereits bekannten Seiten erfolgen. Durch das erneute Abrufen erhöht sich zusätzlich die Abdeckung, da neue Links bei Änderungen einer Seite entdeckt und verfolgt werden können (Olston und Najork 2010, S. 217).

Eine Herausforderung ist die Entscheidung, wann eine Seite erneut abzurufen ist. Sollte keine Sitemap für eine Webseite existieren, kann ein Web Crawler erst nach dem Abruf erkennen, ob sich die Seite verändert hat. Die Entscheidung liegt für jede Seite entweder für oder gegen den Abruf, und somit darin, gegebenenfalls unnötig Ressourcen verwendet zu haben, oder eine veraltete Seite im Repository zu behalten. Eine triviale Strategie, wie die Wiederholung von Abrufen geregelt wird, ist die der *zufallsgesteuerten Auswahl* der Seiten und ein Abruf dieser. Alternativ können *wichtige Seiten* (nach bspw. PageRank- oder Indegree-Score) häufiger abgerufen werden. Eine weitere Alternative ist die *Reduktion von alten Seiten*, wodurch die Seiten, die am längsten nicht besucht wurden, als nächstes abgerufen werden. Ebenfalls zeitabhängig ist die Strategie, Seiten aufgrund ihrer Dauerhaftigkeit

(engl. *longevity*) zu priorisieren und somit abhängig von der geschätzten Änderungsfrequenz abzurufen. Seiten mit seltenen Änderungen sollten dementsprechend selten wiederbesucht werden. Seiten mit häufigen Änderungen dagegen sollten oft wiederbesucht werden. Im Gegensatz zur intuitiven Annahme sollten Seiten, die sich sehr häufig ändern, nicht noch häufiger abgerufen werden. Der höhere Aufwand ist nicht gerechtfertigt, da die Seitenkopie bei der Speicherung im Web Repository bereits veraltet ist (Olston und Najork 2010, S. 219).

4.2.3 Gleichzeitige Maximierung von Abdeckung und Aktualität

In der Literatur gibt es keine einstimmige Meinung darüber, wie eine optimale Balance zwischen Abdeckung und Aktualität geschehen kann (Olston und Najork 2010, S. 216). Olston und Najork nennen dennoch zwei Strategien, die sowohl die Erhöhung der *Abdeckung* als auch der *Aktualität* bei der Priorisierung der Abarbeitungsliste bewirken. Dies ist die Strategie des *WebFountain* Crawlers (Edwards et al. 2001) und die *OPIC*-Strategie (Abiteboul et al. 2003).

Tabelle 6: Strategien mit Abdeckung und Aktualität als Ziel (aus dem Englischen aus Olston und Najork 2010, S. 201)

Technik	Web-Crawler-Metriken		Webseiten-Metriken		
	Abdeckung	Aktualität	Wichtigkeit	Relevanz	Änderungsrate
WebFountain	X	X			X
OPIC	X	X	X		

Die beiden Strategien berücksichtigen jeweils einen unterschiedlichen Faktor, wie in Tabelle 6 dargestellt ist. *WebFountain* berücksichtigt vorrangig die *Änderungsrate* von Webseiten und somit stärker die *Aktualität*. Die *Abdeckung* wird bei dieser Strategie dahingehend berücksichtigt, dass Seiten, die noch nicht heruntergeladen wurden, den Aktualitätswert null besitzen und somit gegenüber anderen Seiten bevorzugt werden. Bei der *OPIC*-Strategie wird vorrangig die *Wichtigkeit* einer Seite berücksichtigt, zudem werden bereits besuchte Seiten in vorgegebenen Zeitabständen erneut aufgerufen (Olston und Najork 2010, S. 216).

Eine Alternative zu den beiden Doppelstrategien ist die Verwendung von zwei Abarbeitungslisten, welche jeweils eine Strategie – Abdeckung oder Aktualität – verfolgen (Cambazoglu und Baeza-Yates 2015, S. 27).

4.2.4 Priorisierungsstrategien im Vergleich

Baeza-Yates et al. (2005, S. 869) haben mehrere der genannten Strategien gegenübergestellt. Dafür wurden Crawl-Durchläufe auf bestehende Datensätze simuliert. Für den Vergleich der Strategien wurden die Metriken *kumulativer PageRank* und *Kendalls* τ verwendet. Der *kumulative PageRank* als Summe der bereits heruntergeladenen Seiten zu bestimmten Crawlzeitpunkten zeigt die Effizienz der Strategien. Unter der Voraussetzung, dass wichtige Seiten zuerst heruntergeladen werden sollen, ist eine Strategie effizienter, wenn der *kumulative PageRank* zu Beginn eine größere Steigung besitzt (siehe Abbildung 10).

Für eine Gegenüberstellung der kompletten Web Crawls wurde zusätzlich der *Durchschnitt des kumulativen PageRanks* für jede Strategie berechnet. *Kendalls* τ ist eine Metrik um die Korrelation sortierter Listen zu erfassen. Sind zwei Listen identisch ist $\tau = 1$, besteht keine Korrelation ist $\tau = 0$ und sind die Listen gegeneinander invertiert ist $\tau = -1$. Die Referenzliste hierfür ist eine nach PageRank-Werten sortierte Liste des gesamten Datensatzes (Baeza-Yates et al. 2005, S. 867 f.).

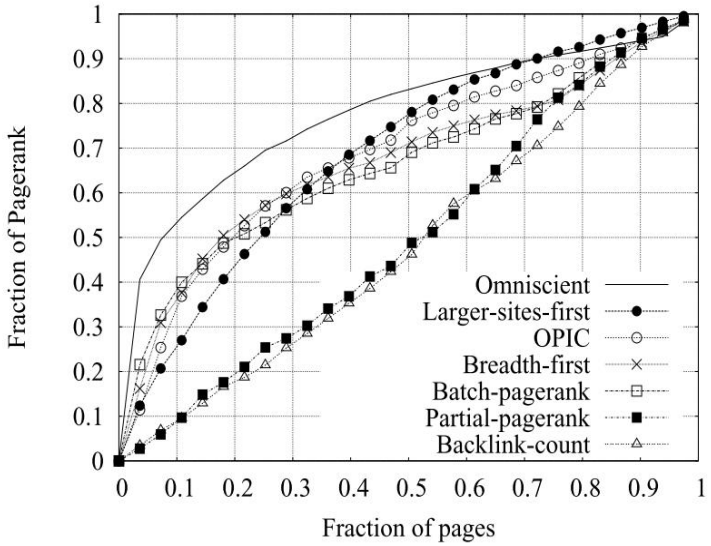

Abbildung 10: Teilergebnis für die Erreichung des PageRanks (Baeza-Yates et al. 2005, S. 869)

Es wurden die Strategien *Breitensuche* (engl. breadth-first search), *Batch-PageRank*, *Indegree*, *larger-sites-first* und *OPIC* gegenübergestellt. Ein allwissender Algorithmus (engl. omniscient), der den PageRank aller Seiten kennt, wurde als bestmöglicher Algorithmus zum Vergleich hinzugefügt. Die Strategien *larger-sites-first* und *OPIC* erreichen dabei die höchsten Werte für den *durchschnittlichen kumulativen PageRank* und für *Kendalls* τ (siehe Tabelle 7). Die Ergebnisse dieser Strategien sind dem Ergebnis der allwissenden Strategie am nächsten. *Larger-sites-first* benötigt im Gegensatz zu *OPIC* weniger Rechenleistung, Speicherressourcen und Datenübertragungsrate. Es wird keine Speicherung und kein Austauschen der Anzahl der Links benötigt (Baeza-Yates et al. 2005, S. 870).

Tabelle 7: Vergleich der untersuchten Priorisierungsstrategien (aus dem Englischen, Baeza-Yates et al. 2005, S. 870)

Strategie	Durchschnittlicher PageRank ± σ	Kendalls τ
Indegree	0.50 ± 0.01	0.0157
Breitensuche	0.64 ± 0.04	0.1293
Batch-PageRank	0.63 ± 0.03	0.1961
OPIC	0.67 ± 0.03	0.2229
Große-Seiten-Zuerst	0.67 ± 0.01	0.2498
Allwissend	0.74 ± 0.04	0.6504

Die Ergebnisse sind für Web-Crawling-Systeme in Größenordnungen von 10-100 Millionen URLs übertragbar. Bei Crawling-Vorgängen von größeren Umgebungen ist es nicht mehr möglich die URLs der Seiten im Hauptspeicher zu lassen. Auch, wenn lediglich die Domains für das gesamte Internet gespeichert werden sollen, besteht eine große Herausforderung. Trotz dieser Schwierigkeit

bezüglich der Datenmenge wird die Speicherung und Verwendung von Informationen aus historischen Daten für die Priorisierung empfohlen (Baeza-Yates et al. 2005, S. 870).

4.3 Partitionierungsstrategien

Als zweite Fragestellung wird die Verteilung der Gesamtliste von einer Steuerungskomponente an viele Fetcher behandelt. Die Verteilung der URLs per REST API ist bereits als Systemvorgabe festgelegt. Je höher die Anzahl der Fetcher desto höher soll auch die Gesamtleistung des Systems sein. Die Leistung kann mit jedem weiteren Fetcher allerdings nur steigen, wenn geeignete Strategien für die Aufteilung der Gesamt-URL-Liste verwendet werden.

Als Vorgabe gilt zudem, dass von Webseiten vorgegebene Verzögerungszeiten aus der `robots.txt` berücksichtigt werden und somit deren Ressourcen geschont werden sollen. Die Steuerungskomponente ist die entscheidende Komponente, in der bestimmt wird, ob der Fokus des Web Crawlers auf der Performanz oder auf der Rücksicht auf fremde Ressourcen liegt (Ueda 2013, S. 75). Des Weiteren werden für das zu entwickelnde System sinnvolle Strategien beschrieben und in einem Ergebnis zusammengeführt.

4.3.1 Verschachtelte Warteschlange

Die wichtigste der gewählten Anforderungen ist die *Schonung von fremden Ressourcen*. Im Folgenden wird daher übergreifend der Ansatz gewählt, URLs zusammenzufassen, die zum gleichen Umfeld gehören. Die Zusammenfassung von in Webseiten enthaltenen Unterseiten wird in Abbildung 11 dargestellt. Darin wird zwischen der Verwendung von Langzeitwarteschlangen für die Webseiten und den gespeicherten Verweisen in Kurzzeitwarteschlangen mit deren Unterseiten unterschieden (Baeza-Yates et al. 2005, S. 866).

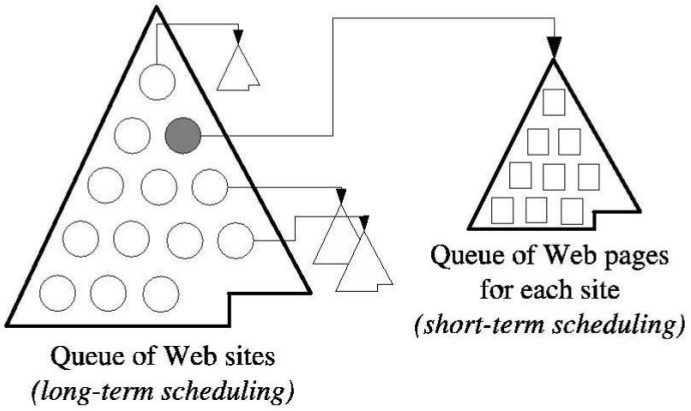

Abbildung 11: Verschachtelte Warteschlangen (Baeza-Yates et al. 2005, S. 866)

Durch diesen Ansatz ist es möglich, die URLs einer Webseite genau einem Fetcher zuzuweisen und gleichzeitige Aufrufe auf eine Webseite von unterschiedlichen Instanzen zu verhindern. Zudem bleibt gewährleistet, dass der Durchsatz in einer Fetcher-Komponente hoch ist, wenn eine Parallelisierung der Langzeitwarteschlangen stattfindet. Die Strategie der verschachtelten Warteschlange wird in den folgenden Unterkapiteln als Grundlage verwendet.

4.3.2 URL-basierte Partitionierung

Die ersten Partitionierungsstrategien orientieren sich an den Informationen, die in der abzurufenden URL enthalten sind. Für eine Partitionierung kann entweder die Top-Level-Domain verwendet werden oder die Information der gesamten Seite.

4.3.2.1 TLD

Ein einfacher Ansatz ist die Erstellung einer Partitionierung basierend auf den Top-Level-Domain-Informationen. Die Informationen werden in diesem Ansatz direkt weiterverwendet. Durch die Aufteilung ist es möglich, Webseiten aus vorgegebenen Ländern konkreten Web Crawlern zuzuweisen (Cho und Garcia-Molina 2002, S. 128).

4.3.2.2 FQDN-Hashing

Die Partitionierung mithilfe einer Hashfunktion erleichtert die Zusammenfassung von Einzelseiten nach dem Kurzzeit- und Langzeitwarteschlangenprinzip. Es wird der komplette Domainname (FQDN) jeder URL in einen Hash-Wert umgerechnet. Dieser Hash-Wert ist zudem Modulo-basiert auf der Anzahl an verfügbaren Fetchern und erlaubt somit sowohl eine einfach zu implementierende als auch eine gleichmäßige Verteilung der Webseiten auf die Fetcher. Durch die exklusive Bearbeitung einer Webseite kann das vorgegebene Crawl Delay berücksichtigt werden.

Zwischen den Partitionen bzw. Fetcher-Komponenten gibt es keine Kommunikation (Exposto et al. 2008, S. 546). Die bei Hash-Funktionen auftretenden Kollisionen werden als sehr unwahrscheinlich eingeschätzt (Cambazoglu und Baeza-Yates 2015, S. 19). Eine Kollision hätte zudem keine negativen Auswirkungen auf die *Schonung fremder Ressourcen*, da die Intervalle zwischen den Aufrufen auf eine Webseite in dieser Weise nicht kürzer werden. Durch die Verlängerung der Wartezeiten bei zwei kollidierenden Webseiten in der gleichen Warteschlange ist jedoch eine Verringerung der *Auslastung* möglich.

Der Nachteil dieser Strategie bezieht sich auf den Anforderungspunkt der Skalierbarkeit und des möglichst geringen Aufwandes für die Steuerungskomponente, bei einer Änderung der Anzahl beteiligter Fetcher. Durch das Hinzufügen oder Entfernen von Fetcher-Komponeten wird eine Neuberechnung des Modulo-basierten Hash-Werts aller Webseiten für alle Fetcher benötigt (Boldi et al. 2004, S. 714).

4.3.2.3 Consistent Hashing

Diese Skalierbarkeit wird durch die Strategie des Consistent Hashing mithilfe eines Hash-Rings ermöglicht. Im Hash-Ring werden die Webseiten sortiert nach ihren Hash-Werten gespeichert. Zudem werden neue Fetcher im gleichen Ring einsortiert, indem ihnen ein Hash-Wert zugewiesen wird. Die Zuweisung der Seiten zu den Fetcher-Komponenten erfolgt im Uhrzeigersinn zum nächsten verfügbaren Fetcher. Abbildung 12 zeigt das Hinzufügen eines weiteren Fetcher C zu den bestehenden beiden. Vor der Ergänzung (i) sind die Webseiten 1 und 2 dem Fetcher A zugewiesen, die Webseiten 3, 4 und 5 dem Fetcher B. Mit dem Hinzufügen von Fetcher C (ii) verändert sich die Zuordnung der Seiten und Fetcher. Die Webseiten 1, 2 und 5 behalten ihrer Zuweisung, die Webseiten 3 und 4 werden dem neuen Fetcher C neu zugewiesen.

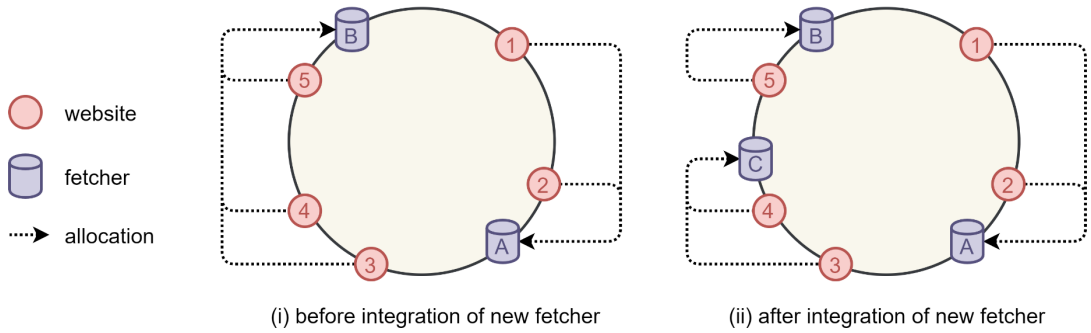

Abbildung 12: Webseitenzuweisung zu Fetcher-Komponenten mit Consistent-Hashing-Strategie (in Anlehnung an Karger et al. 1999, S. 1206)

Die Speicherung der Hashwerte der Webseiten erfolgt bei der Consistent-Hashing-Strategie als Binärbaum und ermöglicht eine schnelle Durchsuchbarkeit nach den Hashwerten der Fetcher (Karger et al. 1999, S. 1206). Webseiten mit sehr vielen Unterseiten stellen allerdings ein Problem dar, da es davon nicht sehr viele gibt. Eine Einordnung einer solchen großen Webseite zu einem Fetcher führt zu einer ungleichmäßigen Verteilung (Nasri und Sharifi 2009, S. 718). Zudem besteht keine Möglichkeit geographische Nähe durch Consistent Hashing abbilden zu können (Malet 2009, S. 22).

Durch Consistent Hashing wäre somit die erste Anforderung der *Schonung fremder Ressourcen* gegeben und eine *hohe Auslastung* durchführbar. Ebenso kann durch Consistent Hashing die Anforderung des geringen Aufwands für die Änderung der Fetcheranzahl erfüllt werden. Die Anforderungen der Zuweisung von Webseiten zu nahen Web Crawlern und die Probleme der wenigen sehr großen Seiten müssen dagegen durch andere Ansätze gelöst werden.

4.3.3 Geo-basierte Partitionierung

Im Gegensatz zu URL-basierten legen geo-basierte Partitionierungsstrategien den Fokus auf die schnelle Abarbeitung von Aufträgen. Dies geschieht durch die Minimierung der Nähe von einer Fetcher-Komponente zu den die Webseiten bereitstellenden Servern. Im Folgenden werden zwei Strategien vorgestellt. Einerseits die Ermittlung der geographischen Nähe über die IP-Adresse und andererseits über die Dauer von einer Anfrage bis zu einer Antwort.

4.3.3.1 Geographische Nähe

Eine Partitionierung von Webseiten, die in der geographischen Nähe des Web Crawlers liegen ist über IP Lookup Services möglich. In Tests von (Exposto et al. 2005) wurden 88 % der vorgegebenen IP-Adressen vom Lookup Service NetGeo[4] gefunden. Die geographischen Orte wurden mit einer Genauigkeit von 42,5 % korrekt zugewiesen (Exposto et al. 2005, S. 58). Wie die Genauigkeit berechnet wurde, wurde allerdings nicht dokumentiert. Sollte ein solcher Service Stuttgart statt Karlsruhe als Ergebnis ergeben und somit fehlerhaft sein, wäre es für den Anwendungsfall der Masterarbeit dennoch ausreichend.

[4] https://www.caida.org/tools/utilities/netgeo/ (Service nicht mehr verfügbar, abgerufen am 29.01.2020)

Der Service NetGeo wird seit 1999 nicht mehr angeboten. Für die Implementierung müsste auf andere, ähnliche Dienste ausgewichen werden. Auf der Webseite von NetGeo werden Alternativen genannt.

4.3.3.2 Paketumlaufzeit

Als Metrik für die Entfernung im Internet zwischen zwei Punkten (z.B. Server und Client) kann die Paketumlaufzeit (engl. Round Trip Time, kurz RTT) verwendet werden.

RTT beschreibt die Zeit für einen kompletten Verbindungsdurchlauf und ist aufgeteilt in die folgenden drei Abschnitte:

1) DNS Auflösung zur IP-Adresse des Servers
2) TCP-Latenz des Three-Way-Handshakes zwischen Fetcher und Server
3) HTTP-Latenz von der `GET`-Anfrage bis zum ersten empfangenen Byte der Antwort

Die Erfassung ist aufwendig, da jeder Server aktiv angefragt und nachverfolgt werden muss. Außerdem ist die Erfassung sehr fehleranfällig, da viele kurzfristige und zufallsbedingte Faktoren die Abfrageergebnisse beeinflussen können (Huffaker et al., S. 2). Daher bietet sich die Verwendung von RRT-basierten Metriken mit historischen Durchschnittswerten an. Ein Vergleich dazu ist in Abbildung 13 dargestellt. Der Median-Wert, der sich aus den RTT-Werten der letzten 24 Stunden zusammensetzt, ist mit über 90 % Erfolgsrate am höchsten. Ebenfalls in der Abbildung sichtbar sind die Abweichungen, die im Laufe eines Tages (abgetrennt durch vertikale Striche) durch die einzelnen Werte (grüne Linie) entstehen.

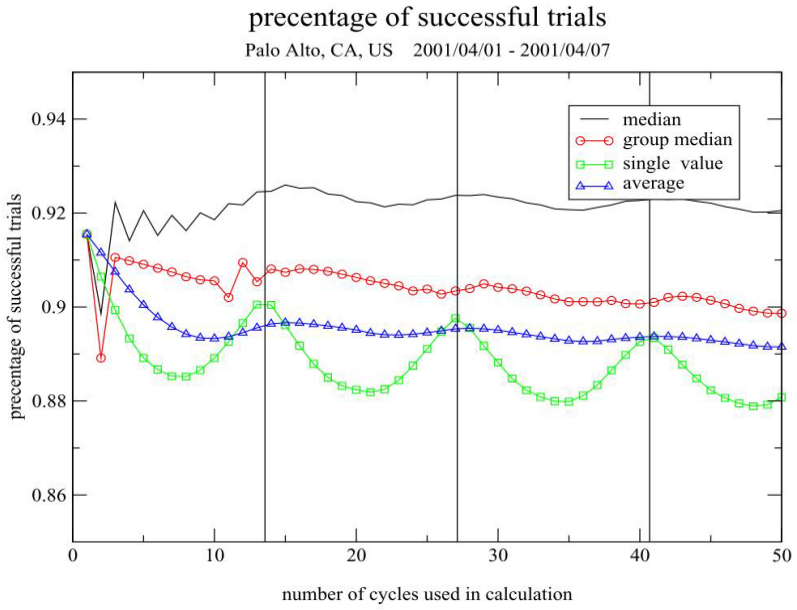

Abbildung 13: Vergleich der Erfolgsraten RTT-basierter Metriken (Huffaker et al., S. 5)

Zwei Nachteile hat diese Strategie: Erstens ist die Metrik der Paketumlaufzeit erst verfügbar, wenn Teile einer Webseite bereits heruntergeladen wurden. Neu entdeckte Unterseiten von Webseiten, die nicht heruntergeladen sind, können somit nicht einem nahen Fetcher zugewiesen werden. Zweitens

reicht ein Wert für die Paketumlaufzeit bei einem verteilten Web-Crawling-System nicht aus. Es müssten die RTT-Werte aller Fetcher für jede Webseite gespeichert werden.

4.3.4 Partitionierungsstrategien im Vergleich

Im Vergleich zur Bearbeitung der URL-Listen eines zentralen Web-Crawling-Systems reduziert sich durch die Verwendung von site-hash-basierten Strategien die Downloadzeit um ca. 40 %, abhängig von der Anzahl der Partitionen. Je mehr Partitionen verwendet werden, desto größer die Reduktion. Durch den Ansatz, die Verteilung auf die geographische Nähe zu optimieren, kann die Downloadzeit um weitere 95 % reduziert werden. Auch hier gilt, dass durch eine höhere Anzahl an Partitionen eine größere Reduktion erreicht wird (Exposto et al. 2005, S. 59). Das vollständige Diagramm ist in Abbildung 14 dargestellt.

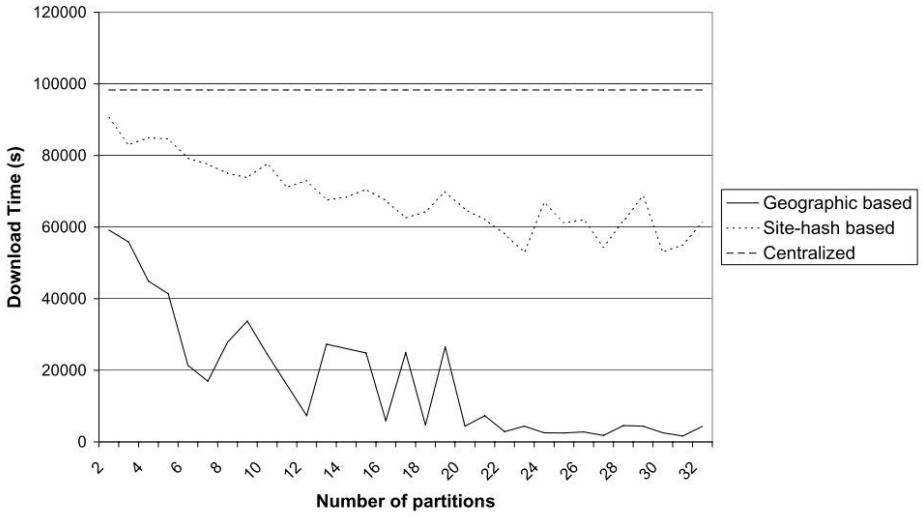

Abbildung 14: Download-Zeiten für Partitionierungsstrategien (Exposto et al. 2005, S. 59)

4.4 Datenstruktur

Die vom Web-Crawling-System abzuarbeitenden URLs werden in der Frontier-Liste abgelegt. Bezüglich der Datenstruktur dieser Liste gibt es unterschiedliche Ansätze.

Die Standardimplementierung einer Priorisierungsliste ist ein Heap, der auf der Festplatte gespeichert wird. Dies ist allerdings nicht performant genug, um die einzelnen URLs in Echtzeit dem Crawler einpflegen und übergeben zu können (Cambazoglu und Baeza-Yates 2015, S. 18). Auch eine Speicherung im Arbeitsspeicher ist aufgrund der benötigten Größe schwierig darzustellen. So würde eine Frontierliste mit 35 Milliarden URLs knapp 4 Terabyte Speicher benötigen (Cui et al. 2018, S. 2). Eine skalierbare Alternative zu einem Heap-Speicher ist die Verwendung vieler FIFO Warteschlangen, die jeweils mit URLs ähnlicher Wichtigkeit befüllt werden. Es werden dabei lediglich die aktuell zu bearbeitenden Warteschlangen im Arbeitsspeicher abgelegt (Cambazoglu und Baeza-Yates 2015, S. 18).

Ein weiterer Ansatz wird für den IRLbot Crawler beschrieben (Lee et al. 2008). Jede Domain erhält ein berechnetes Budget. Bis zu einem vorbestimmten minimalen Budget gibt es für jeden Wert eine eigene Warteschlange. Ist das berechnete Budget einer Domain unter dem minimalen Budget, wird

die Domain in einem Backlog gespeichert. Der Backlog ist im Gegensatz zu den Warteschlangen auf der Festplatte gespeichert. Für sehr große Web Crawler, die Userklicks nicht als Feedback verwenden können scheint dies derzeit die einzige Strategie, abgesehen von einer Skalierung der Server-Cluster (Cui et al. 2018, S. 3 f.).

4.5 Ergebnis der Analyse

Für die beiden zu berücksichtigenden Aspekte Priorisierung und Verteilung der URL-Liste eines Web-Crawling-Systems mit einer zentralen Steuerungskomponente und verteilten Fetcher-Komponenten wurden Strategien aus der wissenschaftlichen Literatur zusammengetragen. Für jede Strategie wurde die Metrik für die Qualität angegeben und durch welche Informationen der abzurufenden Webseiten diese Metrik beeinflusst wird. Als Ergebnis entstehen zwei Listen, die für den Entwurf und die Reihenfolge der Realisierung als Grundlage dienen.

Tabelle 8 listet Priorisierungsstrategien auf, zusammen mit den Metriken und Informationen der jeweiligen Webseiten.

Tabelle 8: Zusammenfassung der gesammelten Priorisierungsstrategien

Priorisierungsstrategie	Web-Crawler-Metrik	Webseiten-Metrik
Breitensuche	Abdeckung	-
Indegree	Abdeckung, Aktualität	Anzahl eingehender Links (Wichtigkeit)
Batch-Pagerank	Abdeckung, Aktualität	Anzahl eingehender Links (Wichtigkeit)
Große-Seiten-Zuerst	Abdeckung	Anzahl Unterseiten
Zufallsgesteuert	Aktualität	-
Änderungsfrequenz	Aktualität	Historische Änderungsfrequenzen
Alte-Seiten-Zuerst	Aktualität	Letztes Abrufdatum
OPIC	Abdeckung, Aktualität	Wichtigkeit + Alter der Seite
WebFountain	Aktualität, Abdeckung	Änderungsfrequenz

In folgender Tabelle 9 sind die in diesem Kapitel gesammelten Partitionierungsstrategien aufgelistet. Die zu optimierende Metrik ist mit dem *Durchsatz* des Gesamtsystems bei allen Strategien gleich. Als verarbeitete Informationen der Webseiten und Unterseiten wird die *URL* verwendet. Für die geobasierten Ansätze wird die *IP* oder die *Paketumlaufzeit* berücksichtigt.

Tabelle 9: Zusammenfassung der gesammelten Partitionierungsstrategien

Partitionierungsstrategie	Web-Crawler-Metrik	Webseiten-Metrik
TLD-Kategorisierung	Durchsatz	URL
FQDN Hash	Durchsatz	URL
Consistent Hashing	Durchsatz	URL
Geographische Nähe	Durchsatz	IP-Adresse
Paketumlaufzeit	Durchsatz	Paketumlaufzeit

5 Entwurf der Steuerungskomponente

Aus den Grundlagen und den Analyseergebnissen erfolgt in diesem Kapitel der Entwurf für die zu erstellende Steuerungskomponente. Dafür werden die Ziele für den Prototyp aus den vorangegangenen Kapiteln zusammengefasst und konkrete Anwendungsfälle beschrieben. Des Weiteren wird der Prototyp in die grundlegende geplante Architektur des Open Web Indexes eingegliedert und das Architekturmodel, das Datenmodell und die Reihenfolge der Entwicklung der einzelnen Strategien beschrieben.

5.1 Ziele der Realisierung

Im Kapitel *Rahmen der Analyse* (Kapitel 4.1, Seite 16 ff.) wurden bereits die Systemvorgaben untersucht und erläutert. Die folgenden drei Tabellen fassen die Auflistung dieser Ergebnisse zusammen.

Tabelle 10 listet die Vorgaben der Fetcher-Komponenten, der Steuerungskomponente und des Gesamt-Systems nochmals auf.

Tabelle 10: Systemvorgaben der einzelnen Komponenten und des Gesamtsystems

Komponente(n)	Vorgaben
Fetcher-Komponente	geographisch verteilt
	ressourcensparend
Steuerungskomponente	zentralisiert
	Zugriff auf Meta-Informationen von URLs
	per REST API ansprechbar
Gesamtes Web-Crawling-System	kontinuierlich & themenunabhängig
	unabhängig von Nutzerabfragen
	unabhängig von Suchergebnisrängen

Des Weiteren werden in Tabelle 11 die zu untersuchenden und gegebenen Metriken aufgelistet.

Tabelle 11: Auswahl der zu untersuchenden Metriken

Web-Crawler-Metriken	Webseiten-Metriken
Abdeckung	Wichtigkeit
Aktualität	Änderungsrate
Durchsatz	

Die Anforderungen und Herausforderungen (in dieser Arbeit kurz: Aufgaben) der Priorisierungs- und Verteilungsstrategien als auch der Steuerungskomponente selbst sind in Tabelle 12 gegenübergestellt. Zudem werden die Möglichkeiten aufgelistet, die eine Optimierung der einzelnen Aufgabe ermöglichen. Des Weiteren wird dargestellt, auf welche Weise eine Messung geschehen kann und welche Informationen der Webseiten verwendet werden können.

Tabelle 12: Aufgaben der Steuerungskomponente

Aufgabe	Optimierungs-möglichkeit(en)	Web-Crawler-Metriken	Webseiten-Metriken
Erreichung hoher Auslastung	Priorisierungs- und/oder Verteilungsstrategien	Abdeckung, Durchsatz	Wichtigkeit
Schonung fremder Ressourcen	Verteilungsstrategien	-	-
Abschätzung der Änderungsrate	Priorisierungsstrategien	Abdeckung	Änderungsrate
Skalierbarkeit	Priorisierungs- und/oder Verteilungsstrategien	Abdeckung, Durchsatz	Wichtigkeit

5.2 Anwendungsfälle

Die Anwendungsfälle gehen von den folgenden Verbindungen von der Steuerungskomponente mit anderen Systemen aus. Eine REST-Schnittstelle erlaubt Verbindungen der Fetcher zur Steuerungskomponente. Zudem ist die Steuerungskomponente über eine Datenbankverbindung mit der Frontier-Datenbank verbunden. Abbildung 15 stellt die aus- und eingehenden Datenflüsse der Komponente dar. Der Verbindungsaufbau erfolgt von einzelnen Fetcher-Komponenten zur Steuerungskomponente und von der Steuerungskomponente zur Datenbank.

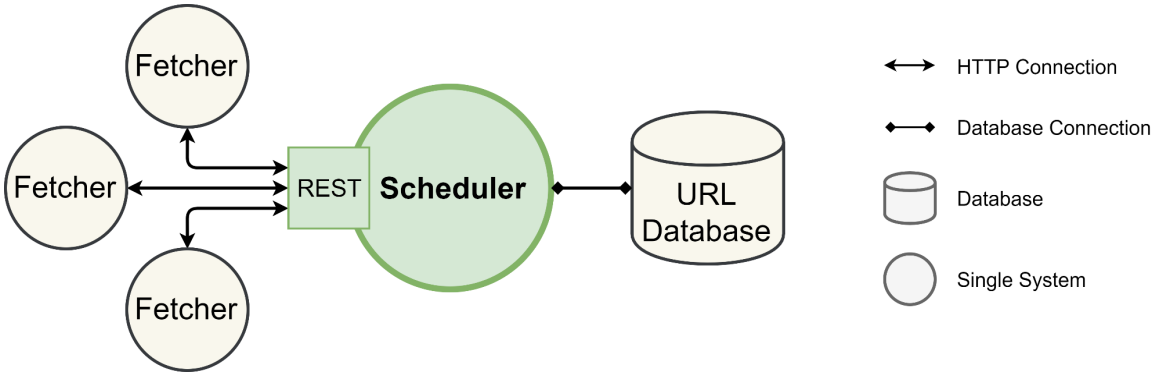

Abbildung 15: Bidirektionale Datenflüsse der Steuerungskomponente

Die Steuerungskomponente ist in zwei Anwendungsfällen enthalten. Dies sind die *Kontoaktionen der Fetcher* und der *Abruf von Abarbeitungslisten*.

5.2.1 Kontoaktionen einer Fetcher-Komponente

Als Kontoaktionen werden im Folgenden die drei Aktionen *erstellen*, *ändern* und *löschen* beschrieben.

Für einen unbekannten Fetcher kann ein *neues Konto* erstellt werden. Die Erstellung erfolgt in vier Schritten:

1) Anfrage per REST-Schnittstelle mit Übermittlung der zwingend erforderlichen Informationen (E-Mail des Besitzers und Name der zu registrierenden Fetcher-Komponente).
2) Prüfung der Daten auf Nichtbestehen aus Sicht der Steuerungskomponente.
3) Generierung einer UUID und Erstellung eines Datenbankeintrags für jeden Fetcher.
4) Antwort auf die Anfrage mit den initial angegebenen und generierten Kontoinformationen.

Mit einem bestehenden Konto können von der Fetcher-Komponente weitere Aktionen per REST API durchgeführt werden. Dies sind die Änderung und Löschung des Kontos und der Abruf von URL-Listen für die Bearbeitung. Für alle Aktionen ist die UUID erforderlich.

Die *Änderung der Kontoinformationen* erfolgt unter Angabe der UUID und der zu verändernden Parameter über die REST-Schnittstelle. Änderbare Informationen sind:

- der Ort der Fetcher-Komponente
- die bevorzugte Top Level Domain und
- die IP-Adresse

Die angegebenen Daten haben Auswirkungen auf die Elemente in der Abarbeitungsliste, in Abhängigkeit von der global verwendeten Strategie.

Des Weiteren kann bei Bedarf ein nicht weiter verwendeter Fetcher aus der Datenbank gelöscht werden. Die *Löschung* erfolgt ebenfalls über die REST-Schnittstelle unter Angabe der UUID.

5.2.2 Abruf einer neuen Abarbeitungsliste

Der Abruf einer neuen URL-Liste erfolgt in drei Schritten. Diese sind die Durchführung der Abfrage vom Fetcher zur Steuerungskomponente, die Kommunikation von der Fetcher-Komponente mit der Datenbank und die Rückgabe der Abarbeitungsliste von der Steuerungskomponente an die Fetcher-Komponente.

Die Anfrage vom Fetcher erfolgt über die REST-Schnittstelle. Bei der Anfrage wird eine gültige UUID benötigt. Weitere optionale Anfrageparameter sind die Anzahl der FQDN-Listen und die maximale Anzahl an URLs pro FQDN-Liste, die zurückgegeben werden soll.

Nachdem eine gültige Anfrage an die Steuerungskomponente übermittelt wurde, erfolgt eine Datenbankabfrage an die Frontier-Datenbank. Diese Abfrage ist abhängig von der Fetcher-Anfrage und den globalen Einstellungen. Bereits von anderen Komponenten reservierte FQDN-Listen werden gefiltert und nicht zurückgegeben. Daraufhin werden die neuen zurückzugebenen FQDN-Listen in einer Reservierungstabelle zusammen mit der UUID der Fetcher-Komponente und dem Ablaufdatum abgespeichert.

Als Rückgabe erhält der anfragende Fetcher eine Liste von FQDN-Listen im JSON-Format. Ab einer Antwortgröße von mehr als 150 Kilobyte wird diese in komprimierter Form übermittelt. Die Rückgabe beinhaltet zudem die Antwort-URL, das Ablaufdatum, die Anzahl der FQDNs und die Gesamtanzahl der URLs.

5.3 Systemumfeld

Im Folgenden wird das Umfeld der zu entwickelnden Steuerungskomponente untersucht. Zu Beginn wird das Architekturkonzept des Open Web Index beschrieben. Darauf aufbauend wird ein Entwurf für Änderungen entwickelt, der eine Eingliederung der Steuerungskomponente in das bestehende OWI-Konzept ermöglicht. Zudem werden einzelne Komponenten im näheren Umfeld und ihre Beeinflussung auf die Steuerungskomponente betrachtet.

5.3.1 OWI Architektur

Das Architekturkonzept wurde bereits im Jahr 2017 von Huss et al. dokumentiert. Die für den hier entstehenden Entwurf relevanten Komponenten des initialen Konzepts sind in Abbildung 16 dargestellt.

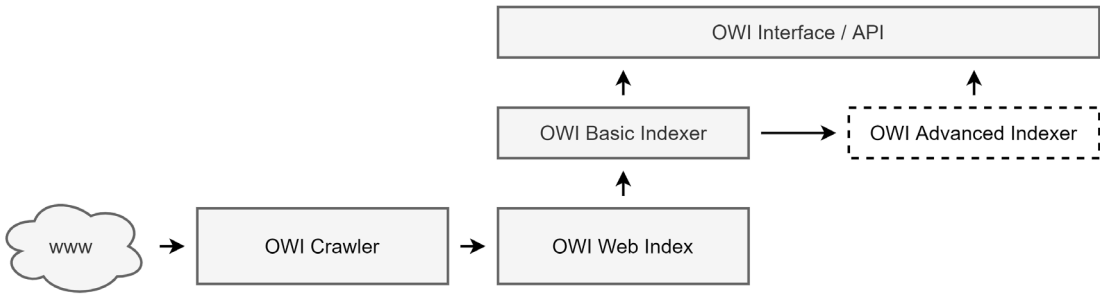

Abbildung 16: Architekturkonzept des Open Web Index (Auszug, nach Huss et al. 2017, S. 5)

Die Abbildung kann als Datenflussdiagramm gelesen werden: Die Informationen der Webseiten gelangen über den OWI Crawler in den OWI Web Index. Die Indexer-Komponenten verarbeiten die Informationen des OWI Web Index und geben diese an das OWI Interface weiter. Das OWI Interface ist der Anlaufpunkt für verschiedene Services, wie zum Beispiel eine Suchmaschinenwebseite.

5.3.2 Integration neuer Komponenten

Um die in den Anwendungsfällen beschriebenen Funktionalitäten in das OWI System einzugliedern, werden, wie in der folgenden Abbildung 17 mit grünen Kästchen dargestellt, die Steuerungskomponente (*Scheduler*), die Datenbank aller bekannten und unbekannten URLs (*URL Database*) und eine Komponente zum Extrahieren der URLs aus den HTML-Seiten (*URL Parser*) in einen Rückfluss vom ehemaligen OWI Web Index zu dem ehemaligen OWI Crawler eingegliedert.

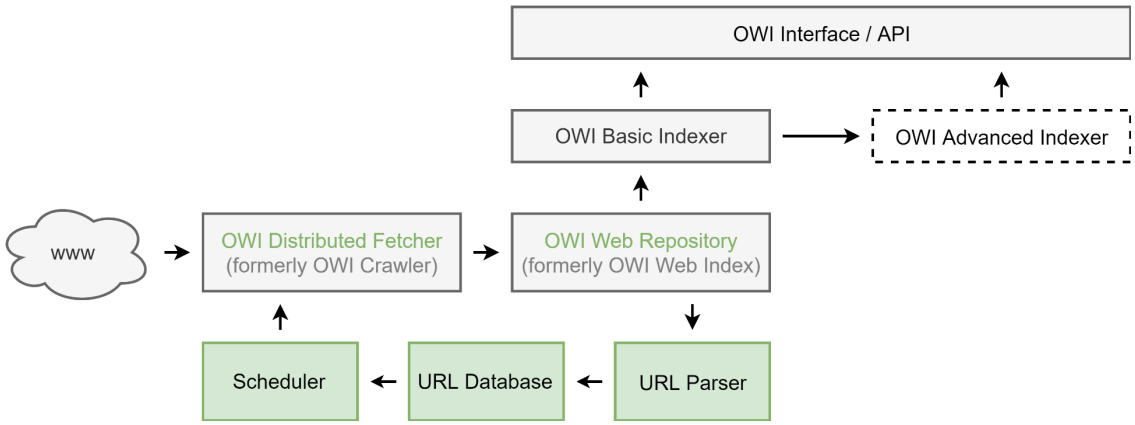

Abbildung 17: Änderungen (grün) der OWI Architektur (eigene Darstellung, angelehnt an Huss et al. 2017, S. 5)

Als weitere Änderung erfolgt eine Umbenennung der bestehenden Komponenten *OWI Crawler* und *OWI Web Index* in *OWI Distributed Fetcher* und *OWI Web Repository*, in Abbildung 17 ist diese Änderung durch grüne Schrift dargestellt. Diese Änderungen beschreiben den in dieser Arbeit verwendeten Anwendungsfall genauer. Es gibt statt einem Crawler viele verteilte Fetcher. Statt eines

Web Indexes, in dem die Informationen bereits verarbeitet sind, werden die Informationen komplett abgespeichert und einerseits von den *Indexer-Komponenten* in einen invertierten Web Index abgelegt und andererseits nach neuen URLs vom Parser im Scheduler System durchsucht und in der URL-Datenbank abgelegt.

5.3.3 Datenfluss im nahen Umfeld

Im Folgenden werden die fünf Komponenten, die in das ursprüngliche Architekturkonzept integriert wurden, vorgestellt. Die Komponenten und ihre Verbindung sind in der nächsten Abbildung (Abbildung 18) isoliert dargestellt. Sie stellen den kleinsten im Umfeld zusammenfassbaren Kreislauf dar. Aufgrund dieses Kreislaufes ist es allerdings nötig alle Komponenten zu betrachten. Das Internet als Quelle der Informationen ist zusätzlich dargestellt.

Als zentrale Komponente agiert für den Ablauf eines Web Crawlers die URL-Datenbank (*URL Database*), in der alle bekannten URLs vorgehalten werden. Eine Steuerungskomponente (*Scheduler)* verwendet diese Liste als Grundlage und verteilt Segmente an die angebundenen *Fetcher*. Ein *Fetcher* ruft die zugewiesenen URLs auf und lädt die Antwortdaten herunter. Im Anschluss oder nach vorgegebenen Zeitabständen sendet er diese Informationen weiter an das *Web Repository*. Im *Web Repository* werden die Daten gespeichert und für weitere Komponenten zur Verfügung gestellt. Mithilfe eines *URL-Parsers* werden die einzelnen Seiten des *Web Repositories* nach URLs durchsucht und Funde zurück an die URL-Datenbank (*URL Database*) gesendet.

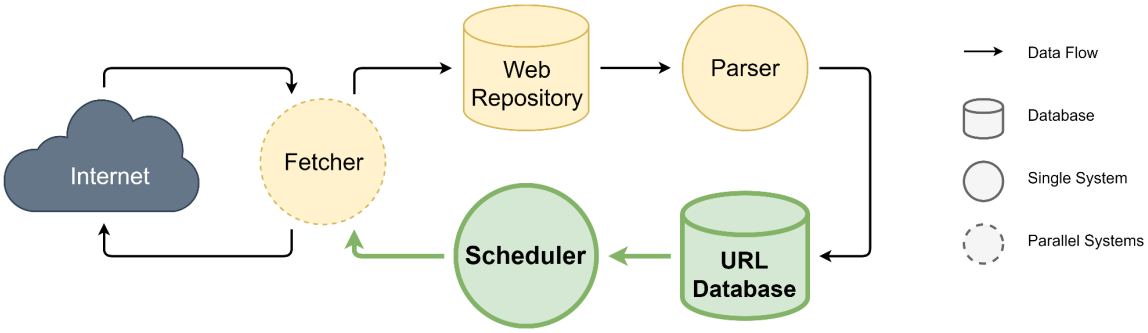

Abbildung 18: Datenfluss zwischen den Komponenten im nahen Systemumfeld (in Anlehnung an Cambazoglu und Baeza-Yates 2015, S. 16, Ueda 2013, S. 75, Lee et al. 2008, S. 5)

Die in grün dargestellte Steuerungskomponente (*Scheduler*) und die URL-Datenbank sind die im Entwurf und Prototyp zu entwickelnden Komponenten. Die URL-Datenbank existierte bisher nicht, ist jedoch zwingend für die Steuerungskomponente erforderlich. Die weiteren gelb dargestellten Komponenten beeinflussen die Entwicklung dieser Komponenten stark und werden dementsprechend berücksichtigt. Des Weiteren sind die Schnittstellen zwischen Steuerungskomponente und den Fetcher-Komponenten und zwischen Steuerungskomponente und URL-Datenbank, dargestellt durch grüne Pfeile, zu entwickeln.

5.3.4 Ausgangspunkt der Entwicklung

Die Betrachtung und Berücksichtigung aller beteiligten Komponenten und ihrer Funktionalitäten ist mit hohem Aufwand verbunden. Dass die Komponenten noch nicht existieren, erhöht den Aufwand nochmals, da dadurch mehrere mögliche Funktionalitäten für jede Komponente existieren können.

Aus diesem Grund wird im folgenden Teil ein geeigneter Ausgangspunkt mit Vereinfachungen und Eingrenzungen der Funktionalitäten der in Abbildung 18 dargestellten Objekte gewählt und beschrieben. Diese Auflistung vereinfacht die Realisierung und beschreibt die Anforderungen an zukünftige Entwicklungsprojekte.

Für die zu entwickelnde *Steuerungskomponente* gilt, wie bereits in der Abgrenzung der Masterarbeit genannt (siehe Seite 4), dass diese selbst nicht verteilt ist. Die dadurch entfallende Kommunikation zwischen einzelnen Steuerungskomponenten ermöglicht die Konzentration auf den Strategievergleich.

Die *URL-Datenbank* ist ebenfalls nicht verteilt. Als vereinfachter Fall wird eine relationale Datenbank verwendet. Sie besteht aus einer Tabelle der Webseiten, gekennzeichnet durch den vollständigen Domainnamen (FQDN) und einer Tabelle mit allen Unterseiten und ermöglicht die Anwendung der verschachtelten Warteschlange, die in Kapitel 4.3.1 beschrieben ist. In den Tabellen sind keine Duplikate erlaubt. Für jede URL ist die Information der zugehörigen `robots.txt`-Datei bereits enthalten und ermöglicht die Abfrage, ob eine URL von einem Fetcher heruntergeladen werden darf. Die aktuelle Abrufverzögerung wird für jede Domain in der FQDN-Tabelle abgespeichert. Eine Komponente zur Seed-Erstellung existiert nicht, FQDNs und URLs können per Datenbankimport in die entsprechenden Tabellen eingefügt werden.

Für die *Fetcher-Komponenten* wird vereinfacht davon ausgegangen, dass diese während der Kommunikation mit der Steuerungskomponente nicht abstürzen können. Bei der Übergabe der heruntergeladenen Seiten an das Web Repository werden zusätzliche Metadaten, wie beispielsweise der Zeitpunkt des Zugriffs, übertragen. Zudem werden lediglich textbasierte Dateien heruntergeladen, das Herunterladen von Rich Media Inhalten und JavaScript-Dateien ist nicht vorgesehen.

5.4 Entwicklungsplan

Im Folgenden werden die gesammelten Anforderungen in Diagrammen zusammengefasst. Zu Beginn wird die Schnittstelle der Steuerungskomponente beschrieben, mit der sich die Fetcher-Komponenten verbinden. Des Weiteren wird das Datenbankmodell der URL-Datenbank erläutert. Abschließend wird Implementierungsreihenfolge der Strategien aufgelistet.

5.4.1 Modell der Schnittstelle

Die Schnittstelle der Steuerungskomponente erlaubt den Zugriff von Fetcher-Komponenten. Die zur Verfügung stehenden Befehle sind in Tabelle 13 aufgelistet. Die Schnittstelle bietet den Zugriff auf Profileigenschaften von Fetcher-Komponenten und den Abruf einer URL-Liste für die weitere Verarbeitung.

Die Zugriffe bezüglich der Profileigenschaften erlauben die Erstellung, Änderung und Löschung einzelner Konten für Fetcher-Komponenten. Für die Erstellung eines Kontos wird eine Kontakt-E-Mail-Adresse und ein Name für die Fetcher-Komponente benötigt. Ist die Erstellung des Kontos erfolgreich wird eine einzigartige ID zurückgegeben, mit der eine Fetcher-Komponente die weiteren Aktionen innerhalb der Schnittstelle ausführen kann. Die sind die Änderung der Profildaten und die Reservierung einer URL-Liste für die Fetcher-Komponente.

Tabelle 13: Übersicht über die REST-Schnittstelle [5]

Kategorie	Pfad	Befehl	Beschreibung	Anfrage Werte	Rückgabe Werte
Profil	/fetchers/	POST	Neues Fetcher-Konto erstellen	Kontakt E-Mail*, Fetcher-Name*, Standort, bevorzugte TLD, IP-Adresse	UUID, Fetcher Hash(es), Kontakt E-Mail, Fetcher-Name, Erstellungsdatum, Standort, bevorzugte TLD,
		PUT	Fetcher-Konto zurücksetzen**	UUID*, Kontakt E-Mail, Fetcher-Name, Standort, bevorzugte TLD, IP-Adresse	
		PATCH	Fetcher-Konto aktualisieren***		
		DELETE	Fetcher-Konto löschen	UUID*	
URL-Liste	/frontiers/	POST	URL-Liste Abrufen und re-servieren	UUID*, FQDN-Anzahl, URLs/FQDN Anzahl	UUID, Rücksende-URL, Ablaufzeitpunkt, FQDN-Anzahl, Gesamt-URL-Anzahl, Liste mit FQDNs, Liste mit URLs

*: zwingend erforderlich **: leere Parameter werden zurückgesetzt ***: leere Parameter werden nicht geändert

Beim Abruf der URL-Liste kann optional die Menge der Webseiten (bzw. Domains) und Unterseiten angegeben werden. Erfolgt dies nicht, werden die Standardwerte der Steuerungskomponente verwendet.

5.4.2 Datenbankmodell

In Abbildung 19 ist das Datenbankmodell dargestellt. Es zeigt die fünf benötigten Tabellen zusammen mit den zugehörigen Fremd- und Primärschlüsseln (FK & PK) und weiteren Attributen. Das Datenbankmodell zeigt die Tabellen *Fetcher*, *FetcherHash Frontier*, *Url* und *FetcherReservation* (im Uhrzeigersinn, beginnend Mitte rechts).

Die Tabelle *Fetcher* speichert die Daten zu allen registrierten Fetcher-Komponenten. Die verfügbaren Attribute sind direkt mit der REST-Schnittstelle verknüpft. Für jeden Fetcher werden, je nach verwendeter Partitionierungsstrategie, Hashwerte generiert. Diese werden in der Tabelle *FetcherHash* abgelegt.

In der Tabelle *Frontier* werden die Informationen zu allen Domains gesammelt. Es können unter anderem die Informationen zur Top-Level-Domain (*tld*), der letzten IP-Adresse (*fqdn_last_ipv4* und *fqdn_last_ipv6*) und der gewünschten Abrufverzögerung (*fqdn_crawl_delay)* abgelegt werden.

Die Beziehungstabelle *FetcherReservation* besitzt einen zusammengesetzten Primärschlüssel, der aus zwei Fremdschlüsseln besteht. Diese referenzieren auf den Primärschlüssel der *Fetcher*-Tabelle und

[5] Online-Version verfügbar unter: http://ec2-18-185-96-23.eu-central-1.compute.amazonaws.com/docs, abgerufen am 19.07.2020.

der *Frontier*-Tabelle. Zudem ist für jede Beziehung das Ablaufdatum (*latest_return*) gespeichert. Einträge mit alten Ablaufdaten werden in regelmäßigen Abständen gelöscht.

Die Tabelle *Url* beinhaltet die Informationen zu den einzelnen Unterseiten der Domains und verweist somit auf den zugehörigen Eintrag in der *Frontier*-Tabelle. Informationen zu URLs sind unter anderem der Zeitpunkt des letzten Abrufs (*url_last_visited*), der Zeitpunkt der Entdeckung (*url_discovery_date*) oder ob die Seite für den automatisierten Abruf zur Verfügung steht (*url_bot_excluded*).

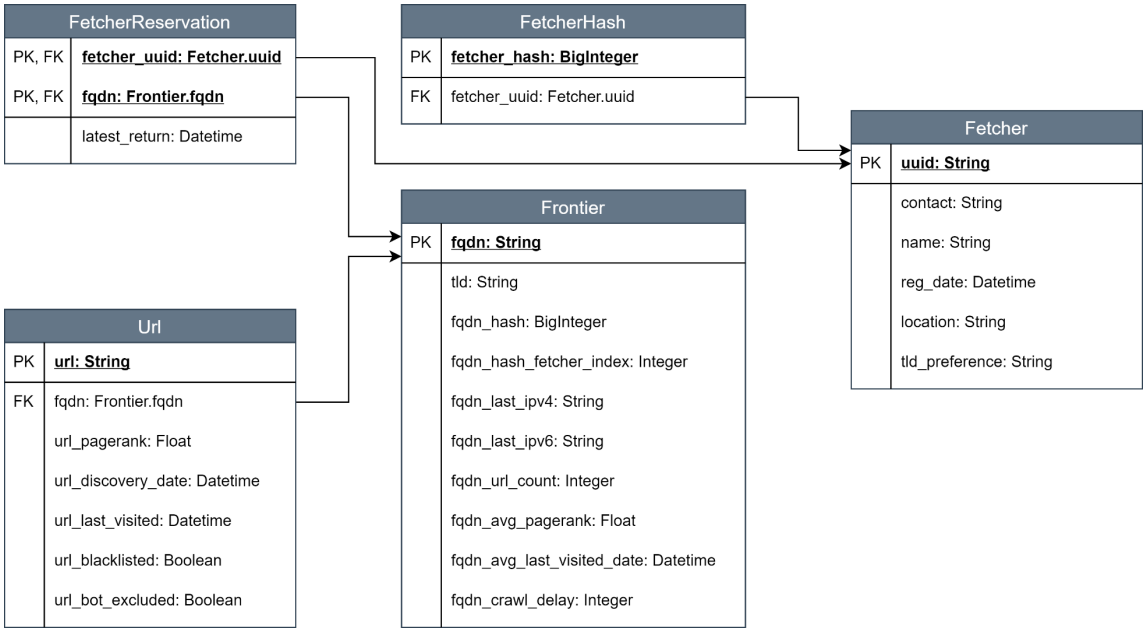

Abbildung 19: Modell der URL-Datenbank

5.4.3 Umsetzungsreihenfolge der Strategien

Im Ergebnis der Analyse (siehe Seite 28) wurden in Tabelle 8 und Tabelle 9 bereits die gesammelten Priorisierungs- und Partitionierungsstrategien aufgelistet. Zu den aus der Literatur gesammelten Strategien wurde als Gegensatz zur *Alte-Webseiten-Zuerst*-Strategie, die *Neue-Webseiten-Zuerst*-Strategie hinzugefügt. Für die Realisierung wurden die Strategien nach Wichtigkeit für die Thesis priorisiert und eine Aufwandsschätzung vorgenommen. Aus diesen beiden Werten wurde eine Reihenfolge gewählt, die in der folgenden Tabelle 14 aufgelistet ist.

Für jede Strategie wird unterteilt, welcher Art sie ist und ob sie als Langzeitstrategie oder Kurzzeitstrategie verwendet werden kann. Langzeitstrategien betreffen FQDN-Listen, Kurzzeitstrategien betreffen die zugehörigen URL-Listen. Für Langzeitstrategien sind sowohl Partitionierungs- als auch Priorisierungsstrategien aufgelistet, für Kurzzeitstrategien lediglich Priorisierungsstrategien.

Tabelle 14: Umsetzungsreihenfolge der Priorisierungs- und Partitionierungsstrategien

Gewählte Reihenfolge	zwingend / optional	Strategieart	Langzeitstrategie	Kurzzeitstrategie	Priorität	Geschätzter Aufwand	Web-Crawling Metrik	Web-Seiten Metrik
1	zwingend	Priorisierung	Random	Random	5	0	-	-
2		Partitionierung	Top-Level-Domain	-	5	1	Durchsatz	URL
3		Priorisierung	Große-Webseiten-zuerst	-	5	1	Abdeckung	Anzahl Unterseiten
4		Priorisierung	Kleine-Webseiten-zuerst	-	5	1	Abdeckung	Anzahl Unterseiten
5		Priorisierung	Alte-Webseiten-zuerst	Alte-Unterseiten-zuerst	5	5	Aktualität	Letztes Abrufdatum
6		Priorisierung	Neue-Webseiten-zuerst	Neue-Unterseiten-zuerst	5	2	Aktualität	Letztes Abrufdatum
7		Partitionierung	FQDN Hash	-	4	5	Durchsatz	URL
8		Partitionierung	Consistent Hashing	-	4	10	Durchsatz	URL
9		Priorisierung	Avg. PageRank (generiert)	PageRank (generiert)	3	2	Abdeckung, Aktualität	Wichtigkeit
10	optional	Priorisierung	Avg. Changerate	ChangeRate	3	7	Aktualität	Historische Änderungsfrequenz
11		Priorisierung	Breadth-First-Search	-	2	1	Abdeckung	Erfassungsdatum
12		Priorisierung	Avg. PageRank (berechnet)	PageRank (berechnet)	3	9	Abdeckung, Aktualität	Wichtigkeit
13		Priorisierung	Avg. InDegree	InDegree	3	10	Abdeckung, Aktualität	Wichtigkeit
14		Priorisierung	Avg. OPIC	OPIC	3	1	Abdeckung, Aktualität	Wichtigkeit + Alter der Seite
15		Partitionierung	Round-Trip-Time	-	3	6	Durchsatz	Paketumlaufzeit (Round Trip Time)

Im Prototyp soll eine beispielhafte Auswahl an Strategien implementiert werden. Dafür werden die Strategien in zwingend und optional kategorisiert. Neun Strategien sollen zwingend entwickelt werden, weitere sechs Strategien sind optional in ihrer Umsetzung.

6 Realisierung der Steuerungskomponente

Es wurden für die Entwicklung verschiedene Technologien und Methoden ausgewählt und in diesem Kapitel dargestellt. Des Weiteren werden die verwendete Infrastruktur und Python-Bibliotheken detailliert beschrieben. Abschließend werden die finalen Systeminformationen zusammengefasst. Ziele für das Endprodukt der Entwicklung sind, neben der Erstellung einer funktionierenden Software, die einfache Konfigurierbarkeit und die einfache Anpassbarkeit für zukünftige Nutzer.

6.1 Werkzeuge und Dienste für die Entwicklung

Als Programmiersprache wird Python eingesetzt. Python ist universal einsetzbar, besonders gut lesbar, beinhaltet eine umfangreiche Standardbibliothek und für weitere Zwecke eine einfache Einbindung von Bibliotheken und Frameworks von Drittanbietern.

Die Entwicklungsumgebung mit den meisten unterstützenden Funktionalitäten ist zum gegenwärtigem Zeitpunkt die Software *PyCharm* von *JetBrains*[6], welche in der *Professional Edition 2020.1* vorliegt. PyCharm bietet Unterstützung für die im folgenden beschriebenen Dienste: Versionierung mit *GitHub*, Virtualisierung mit *Docker*, Funktionstest über *pytest* und die Überprüfung der Qualität des Programmiercodes über *Travis-CI*, *SonarCloud* und *CodeCov*.

Die Versionierung erfolgt über den Dienst von GitHub[7]. Mithilfe der Versionsverwaltung ist es möglich mit mehreren Computern an dem Projekt zu arbeiten, eine Absicherung gegen Verlust des Programmiercodes zu erstellen und fehlerhafte Entscheidungen, die spät entdeckt werden, rückgängig zu machen. Des Weiteren können in GitHub weitere Dienste angebunden werden, die starten, wenn eine neue Version in das Projekt geladen wird. Für dieses Projekt angebundene Dienste sind die Erstellung von Docker-Containern und die Überprüfung der Code-Qualität.

Die Erstellung eines neuen Virtualisierungs-Containers erfolgt automatisiert nach jedem Hochladen einer neuen Version in das GitHub-Projekt. Geschieht die Erstellung ohne Fehler, wird der Container als neue Version in *Docker Hub*[8] bereitgestellt. Zusätzlich wird eine Qualitätskontrolle durchgeführt. Der angebundene Dienst *Travis-CI*[9] wird ebenfalls bei jedem Hochladen einer neuen Version ausgeführt. Der Dienst lässt alle erstellten Tests prüfen und stößt die Aktualisierung der Dashboards von *SonarCloud*[10] und *CodeCov*[11] an. Das Dashboard von SonarCloud erstellt eine Übersicht über neu erkannte Schwachstellen und Sicherheitslücken, über die Anzahl von duplizierten Codefragmenten und gibt eine Kennzahl für die Wartungsfreundlichkeit des Programmcodes aus. Das Dashboard von CodeCov ermittelt die Abdeckung der durch Tests geprüften Codefragmente und gibt Hinweise an welchen Stellen noch Tests zu erstellen sind.

Der Zusammenhang und die Reihenfolge der manuell und automatisiert durchgeführten Einzelschritte zur Bereitstellung einer neuen Version wird in Abbildung 20 verdeutlicht. Am Ende jedes

[6] https://www.jetbrains.com
[7] https://github.com
[8] https://hub.docker.com
[9] https://travis-ci.org
[10] https://sonarcloud.io
[11] https://codecov.io

fehlerfreien Durchlaufs steht die Aktualisierung des verwendeten Docker-Containers innerhalb der AWS EC2 Instanz.

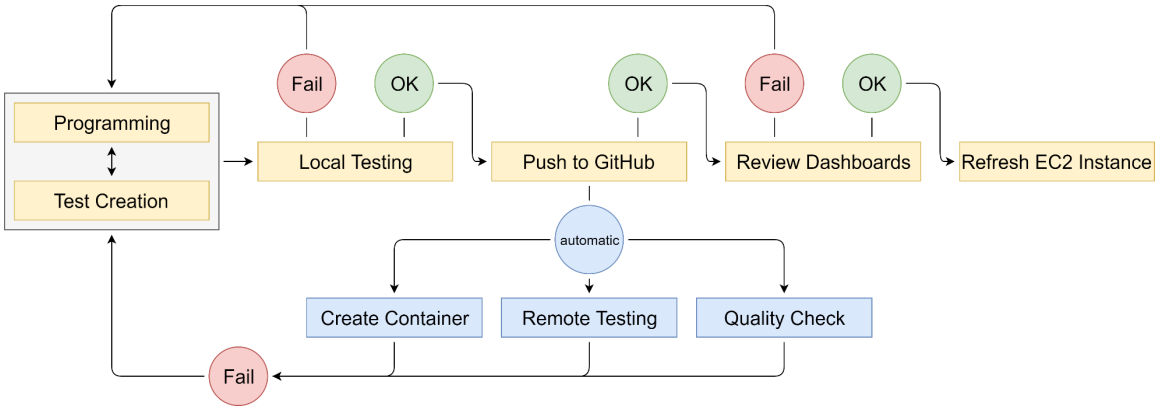

Abbildung 20: Zyklus eines Entwicklungsschrittes

6.2 Verwendete Infrastruktur

Als universaler Cloud-Dienst wurden *Amazon Web Services*[12] (abgekürzt: AWS) des Unternehmens Amazon ausgewählt. Bei der Auswahl der Dienste wurde darauf geachtet, dass diese in gleicher Verfügbarkeit bei anderen Cloud-Anbietern ebenfalls zur Verfügung stehen.

Für die Steuerungskomponente wurde ein virtueller Server gewählt, der Service nennt sich *Elastic Compute Cloud* (abgekürzt: EC2). Der Server verfügt über einen virtuellen Prozessor, 1 GB Arbeitsspeicher und 8 GB Festplattenspeicher. Als Betriebssystem wurde *Ubuntu Server 18.04 LTS* gewählt. Auf dem Server musste lediglich der Service Docker installiert werden, um die bereitgestellten Container verwenden zu können.

Die URL-Datenbank wurde als virtuelle Datenbank erstellt, der Service dazu wird *Relational Database Service* (abgekürzt: RDS) genannt. Als Datenbank wurde PostgreSQL 11.6 gewählt. Die Datenbank besitzt initial 20 GB Speicher und wird bis zu einem Bedarf von 1 TB automatisch vergrößert.

Beide Services werden in dieser Konfiguration als kostenlose Services für 750 Stunden pro Monat angeboten.[13]

6.3 Python Erweiterungen

Die Programmiersprache Python setzt neben ihrer umfangreichen Standardbibliothek auf die Erstellung von Bibliotheken und Frameworks der Python-Gemeinschaft. Diese lassen sich einfach in neue Projekte einbinden. Für die Steuerungskomponente wurden die folgenden Bibliotheken importiert:

[12] https://aws.amazon.com

[13] Kostenlos in den ersten 12 Monaten nach Registrierung (siehe: https://aws.amazon.com/de/free/).

pytest, *FastAPI, Starlette, pydantic, SQLAlchemy, Psycopg* und *xxhash*. Die fünf erstgenannten Bibliotheken sind über die Bestimmungen der *MIT-Lizenz*[14] lizenziert, *Psycopg* wird nach den Bestimmungen der *GNU Lesser General Public License*[15] lizenziert und die Bibliothek *xxhash* ist nach den Bestimmungen der *BSD License*[16] lizenziert. Die Bibliotheken stehen somit für die Verwendung in diesem Projekt zur Verfügung. Die Verwendung von *xxhash* wird aufgrund der Lizenz auf der GitHub-Seite genannt und der Quellcode verlinkt.

Die Bibliothek *pytest* (Krekel 2020) wurde bereits im Kapitel der verwendeten Werkzeuge genannt. Sie dient zur Überprüfung der im Projekt erstellten Funktionen. Die Funktionen werden mit Grenzwerten aufgerufen und die Ergebnisse validiert. Dies erlaubt einerseits die Überprüfung, ob einzelne Funktionen ohne Fehler lauffähig sind und andererseits, ob die Rückgabewerte den Erwartungen entsprechen.

Für die Datenbankverbindung und die Abbildung der Datenbankeinträge auf die im Projekt verwendeten Datenobjekte wird die Bibliothek *SQLAlchemy* (Bayer 2020) eingesetzt. Als Adapter für die Verbindung zur PostgreSQL-Datenbank wird die Bibliothek *Psycopg* (Varrazzo 2020) verwendet.

Das Micro-Framework *FastAPI* (Ramírez 2020) wird für die Bereitstellung der Endpunkte der REST API verwendet. Die dazugehörige OpenAPI Dokumentation wird automatisiert generiert. FastAPI setzt auf dem Micro-Framework *Starlette* (Encode OSS 2020) und der Datenvalidierungsbibliothek *pydantic* (Samuel Colvin 2020) auf.

Starlette bietet an, mithilfe eines TestClients die REST-API innerhalb von (py-)Tests zu überprüfen. Durch die Erstellung von Datenmodellen der Bibliothek pydantic können die Datentypen der eingehenden JSON-Daten validiert werden. Dabei ist es auch möglich Datentypen die nicht zu den Standarddatentypen gehören zu validieren, wie zum Beispiel E-Mail-Adressen, URLs oder UUIDs.

Vom Autor von FastAPI wird außerdem ein Docker-Image bereitgestellt, in dem lediglich die benötigten Anwendungen installiert sind.[17] Dieses Docker-Image bildet die Grundlage für das Docker-Image der Steuerungskomponente. Es beinhaltet bereits die Konfiguration für die Serveranwendungen basierend auf dem leichtgewichtigem Asynchron-Server-Gateway-Interface (kurz ASGI) *Uvicorn*[18] und dem HTTP Server *Gunicorn*[19].

Für die Erzeugung der Hash-Werte wurde die Python-Bibliothek *xxhash (Du 2020)* verwendet, welche als Anbindung an den offiziellen, in C geschriebenen, xxHash-Algorithmus innerhalb von Python-Programmen aufgerufen werden kann. Diese ermöglicht die effiziente Erstellung[20] eines Hash-Wertes für die Domains (FQDNs) und die Fetcher-Komponenten.

[14] https://opensource.org/licenses/MIT
[15] https://www.gnu.org/licenses/lgpl-3.0.html
[16] https://www.freebsd.org/copyright/freebsd-license.html
[17] https://github.com/tiangolo/uvicorn-gunicorn-fastapi-docker
[18] https://www.uvicorn.org
[19] https://gunicorn.org
[20] Gegenüber anderen Algorithmen, siehe Benchmark unter http://www.xxhash.com/

6.4 Implementierte Strategien

Alle zwingend zu entwickelnden Strategien wurden innerhalb der Steuerungskomponente zur Verfügung gestellt. Somit stehen im Prototyp drei Partitionierungsstrategien und sechs Priorisierungsstrategien als Langzeitstrategie zur Verfügung. Zudem kann aus vier Priorisierungsstrategien als Kurzzeitstrategie ausgewählt werden. In der Übersicht der umgesetzten Strategien in Tabelle 15 sind diese aufgeführt.

Alle Strategien haben gemeinsam, dass die Grundlage die FQDN- und die URL-Tabelle bilden und eine Ergebnisliste durch SQL-Abfragen gefiltert und sortiert zurückgegeben wird. Außer der zufallsbasierten Priorisierungsstrategie sind alle Strategien deterministisch.

Tabelle 15: In der Steuerungskomponente verfügbare Strategien

Nr.	Strategieart	Langzeitstrategie	Kurzzeitstrategie
1	Priorisierung	Zufallsgesteuert	Zufallsgesteuert
2	Partitionierung	Top-Level-Domain	-
3	Priorisierung	Große-Webseiten-zuerst	-
4	Priorisierung	Kleine-Webseiten-zuerst	-
5	Priorisierung	Alte-Webseiten-zuerst	Alte-Unterseiten-zuerst
6	Priorisierung	Neue-Webseiten-zuerst	Neue-Unterseiten-zuerst
7	Partitionierung	FQDN-Hash	-
8	Partitionierung	Consistent Hashing	-
9	Priorisierung	vorgegebener PageRank-Mittelwert	vorgegebener PageRank

7 Erprobung der Strategien

Die implementierten Strategien werden nun auf geeignete Art und Weise miteinander verglichen. Das Kapitel beginnt mit der Methodik, die für die Erprobung angewandt wurde, im Anschluss werden die Testfälle beschrieben und die ermittelten Ergebnisse gegenübergestellt und interpretiert. Das Kapitel endet mit einer Beschreibung der Grenzen und der Auslastung der Steuerungs- und Fetcher-Komponenten.

7.1 Methodik des Testsystems

Um möglichst wiederholbare Vergleiche durchführen zu können, wurden zusätzliche Komponenten eingeführt und ein Testvorgang entwickelt, wodurch verschiedene Fälle mit jeweils gleicher Startsituation simuliert werden können.

7.1.1 Übersicht über das Testsystem

Das Testsystem besteht aus den in Abbildung 21 dargestellten Komponenten. Grün dargestellt sind die bereits entwickelte Steuerungskomponente mit REST-Schnittstelle und die URL-Datenbank.

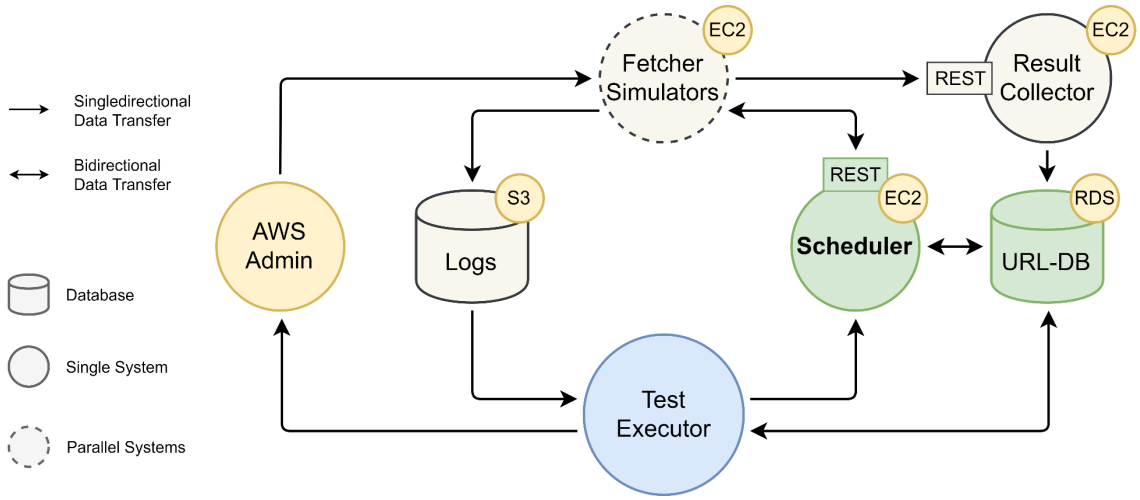

Abbildung 21: Übersicht der Komponenten innerhalb des Testsystems

Hinzugefügt wurde eine Fetcher-Simulator-Komponente, die mithilfe der AWS Admin Konsole beliebig oft gestartet werden kann und mit eigener IP-Adresse im Internet agieren kann. Zudem wurde eine Komponente als Endpunkt für die Ergebnislisten der Fetcher-Simulatoren erstellt, die im weiteren Verlauf der Arbeit als Ergebnissammler (*engl. Result Collector*) bezeichnet wird. Diese wurde synchron zur Steuerungskomponente mit einer REST-Schnittstelle ausgestattet.

Die Log-Dateien der Fetcher-Simulatoren werden in einer zentralen Dokumentendatenbank abgelegt. Mithilfe des lokal laufenden Teststeuerungsprogramms (*Test Executor*, blau dargestellt) können die Testfälle eingestellt, ausgeführt, wiederholt und die Ergebnisse gesammelt werden.

Die neu erstellten EC2 Instanzen (Fetcher und Ergebnissammler) besitzen mit einem virtuellen Prozessor, 1 GB Arbeitsspeicher und 8 GB Festplattenspeicher die gleichen Systemspezifikationen, wie die bereits existierende Steuerungskomponente. Das Teststeuerungsprogramm läuft auf lokalen Computern mit unterschiedlichen Spezifikationen.

7.1.2 Erweiterung der Funktionalitäten der Steuerungskomponente

Bereits während der Erstellung der Steuerungskomponente und der URL-Datenbank erhielt die Steuerungskomponente zusätzliche Funktionalitäten. Diese wurden für die Erstellung des Testsystems als Grundlage genommen und erweitert. Eine Übersicht über die Entwicklerwerkzeuge gibt der Einblick in die erweiterte REST-Schnittstelle der Steuerungskomponente, welche in der folgenden Tabelle in Auszügen aufgelistet ist. Die vollständige REST-Schnittstelle ist in *Anhang B* dokumentiert.

Tabelle 16: Auszug der Entwicklerwerkzeuge innerhalb der REST-Schnittstelle

Pfad	Befehl	Beschreibung	Anfrage Werte	Rückgabe Werte
/fetchers/	GET	Alle Fetcher auflisten	-	Liste mit Fetcher-Komponenten
/database/	POST	Generieren von Beispieldaten	Fetcher Anzahl, FQDN Anzahl, Anzahl URLs pro FQDN,	-
	DELETE	komplette oder teilweise Löschung	Fetcher Hashes, Fetchers, URLs,	
/stats/	GET	Statistiken der Datenbank zurückgeben	-	Anzahl Fetcher, Anzahl FQDNs, Anzahl URLs, Durchschnittliches Alter der URLs, Anteil besuchter URLs
/urls/random/	GET	Zufallsgenerierte URL(s) aus Datenbank zurückgeben	Anzahl gewünschter URLs, FQDN	Liste mit URLs
/settings/	GET	Fetcher-Einstellungen abrufen		siehe PUT /settings/
	PUT**	Einzelne Werte in den Einstellungen ändern	Logging Modus, Crawling Speed Faktor, Standardwert für fehlende Crawl Delays, Anzahl paralleler Prozesse, Anzahl paralleler Fetcher,	

**: leere Parameter werden zurückgesetzt

7.1.3 Einzelschritte des Testvorgangs

Ein Testvorgang besteht aus acht Schritten. Schritt 1 und 2 werden einmalig ausgeführt, Schritt 3 bis 8 werden in Abhängigkeit der Anzahl der Testfälle ausgeführt. Abbildung 22 zeigt die Ausführungsreihenfolge innerhalb der Testarchitektur.

Ein Testvorgang beginnt mit der Anpassung der drei verschiedenen Einstellungspunkte (Schritt 1):

- Einstellungen des Testprojekts
- Einstellungen der Beispieldatenbank
- Einstellungen der zu testenden Einzelfälle

Notwendige Einstellungen bezüglich des Testprojektes sind ein aussagekräftiger Name und die Anzahl der Wiederholungen. Für die Beispieldatenbank kann die Anzahl der initial generierten Domains und die Anzahl der Unterseiten pro Domain eingestellt werden.

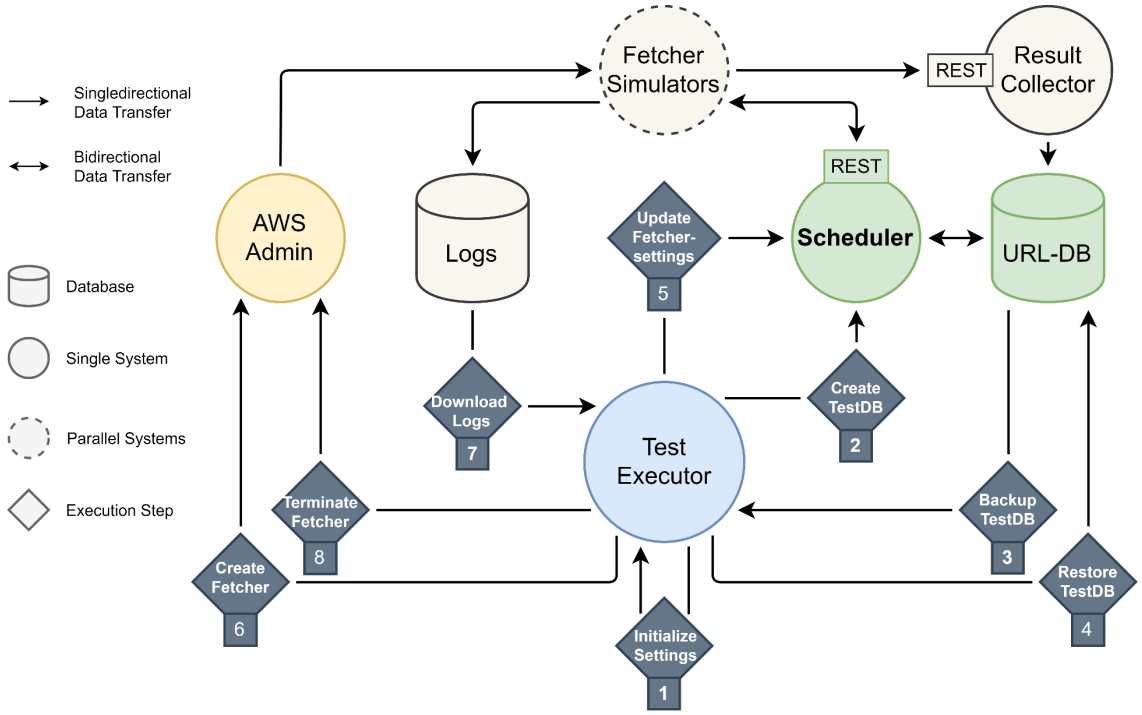

Abbildung 22: Komponenten der Testarchitektur und Reihenfolge der ausgeführten Aktionen

Die zu testenden Einzelfälle können durch die Einstellungen für die Fetcher-Simulatoren (siehe Auszug in Tabelle 17) definiert werden. Es kann beispielsweise die Zahl der zu verwendenden parallelen Prozesse eines Fetcher-Simulators und die Anzahl der parallel laufenden Fetcher-Simulatoren eingestellt werden. Die gesamten Einstellungsmöglichkeiten können in *Anhang C* betrachtet werden.

Tabelle 17: Einstellungen für Fetcher-Simulatoren (Auszug)

Eigenschaft	Beschreibung
`parallel_process`	Anzahl der im Fetcher ausgeführten parallelen Prozesse
`parallel_fetcher`	Anzahl der parallel zu startenden Fetcher-Instanzen
`iterations`	Anzahl der Wiederholungen innerhalb einer Fetcher-Instanz
`fqdn_amount`	Anzahl der pro Iteration maximal abzurufenden Domain-Listen
`url_amount`	Anzahl der pro Domain-Liste maximal abzurufenden URLs
`long_term_prio_mode`	Strategie für die Sortierung der Domain-Listen
`long_term_part_mode`	Strategie für die Aufteilung der Domain-Listen
`short_term_prio_mode`	Strategie für die Sortierung der URLs

In der Teststeuerung lässt sich für jede Fetcher-Einstellung eine Liste an Werten angeben. Die Teststeuerung generiert alle Permutationen und führt diese ab Schritt 3 nacheinander aus.

Nach den Anpassungen der Einstellungen erfolgt die Erstellung der Testdatenbank in Schritt 2. Die Einstellungen für die Testdatenbank werden dafür an die Steuerungskomponente (Scheduler) gesendet. Die Steuerungskomponente generiert daraus einen Beispieldatensatz und sendet diesen an die

URL-Datenbank. Nach erfolgreicher Übermittlung wird eine Sicherung der FQDN-Tabelle und der URL-Tabelle zum Teststeuerungsprogramm transferiert.

Nach den vorbereitenden Maßnahmen der ersten drei Schritte, werden die folgenden Schritte für jeden Testfall jeweils einmal ausgeführt:

4) Testdatenbank zurücksetzen
5) Fetcher-Einstellungen für aktuellen Testfall aktualisieren
6) Fetcher-Simulator(en) starten
7) Warten und Download der Log-Dateien
8) Fetcher-Simulatoren beenden

Die Testdatenbank wird in Schritt 4 zurückgesetzt, indem die in Schritt 2 erstellten Sicherungsdateien in die URL-Datenbank zurückgespielt werden. Daraufhin werden die Einstellungen für den aktuellen Testfall über die REST-Schnittstelle der Steuerungskomponente in der Datenbanktabelle *FetcherSettings* gespeichert. Im Anschluss wird ein Befehl an die AWS-Admin-Konsole gesendet, wodurch die im Testfall definierte Anzahl an Fetcher-Simulatoren gestartet wird. Beim Start ruft die Fetcher-Komponente die aktuellen Fetcher-Einstellungen ebenfalls über die REST-Schnittstelle der Steuerungskomponente aus der Datenbank ab.

Die Log-Dateien mit der Auswertung der einzelnen Simulationen werden von den Fetcher-Simulatoren in einem zentralen Log-Speicher abgelegt. Das Teststeuerungsprogramm sammelt diese Log-Dateien nach jedem Testfall und verarbeitet die darin von den Fetcher-Simulatoren gespeicherten Ergebnisse. Nach dem Transfer der Log-Dateien werden die Fetcher-Simulatoren über die AWS-Admin-Konsole zum Löschen freigegeben.

Am Ende eines Testprojektes wird von der Teststeuerung eine CSV-Datei mit der Zusammenfassung aller Testfälle generiert. Zu jedem Testfall werden die folgenden Informationen gespeichert:

- die Daten der verwendeten Fetcher-Einstellungen
- die Zeiten, die der Fetcher für einzelne Arbeitsschritte benötigt hat
- die Statistiken der Datenbank am Ende des Testvorgangs

Die Attribute der Fetcher-Einstellungen wurden bereits in Tabelle 17 aufgelistet. In den beiden folgenden Tabellen werden die Zeiten der Arbeitsschritte und der Statistiken der Datenbank beschrieben. Tabelle 18 listet die Zeitmessungen innerhalb einer Fetcher-Komponente auf. Die Zeitmessungen werden für jede Iteration der Komponente erfasst.

Tabelle 18: Messwerte einzelner Iterationen

Attribut	Beschreibung
`iter_load_duration`	Dauer der Übertragung der URL-Liste von der Steuerungskomponente zur Fetcher-Komponente, Messung in Sekunden (z.B. 0,187 s)
`iter_fetch_start`	Startdatum und -zeit des Fetch-Vorgangs eines Testfalles (z.B. 2020-07-11 18:52:03,277)
`iter_fetch_duration`	Dauer der Simulation eines Testfalles. Messung in Sekunden (z.B. 4,261 s)
`iter_fetch_cpu_time`	Zeit der Prozessoraktivität während der Simulation eines Testfalles. Messung in Sekunden (z.B. 0,985 s)
`iter_submit_duration`	Dauer der Übertragung zur Komponente, die die Ergebnisse sammelt. Messung in Millisekunden (z.B. 0,051 s)

Abschließend zu jeder `Iteration` wird per REST-Schnittstelle die Datenbankstatistik abgerufen. Es werden Informationen zu den Anzahlen und weitere berechnete Werte zurückgegeben (siehe Tabelle 19).

Tabelle 19: Statistiken der Datenbank

Attribut	Beschreibung
`db_fetcher_amount`	Anzahl Fetcher
`db_frontier_amount`	Anzahl FQDNs
`db_url_amount`	Anzahl URLs,
`db_avg_freshness`	Durchschnittliches Alter der URLs,
`db_visited_ratio`	Anteil besuchter URLs

7.1.4 Anpassung der angewendeten Metriken

Im Folgenden werden die für die Vergleiche verwendeten Metriken beschrieben. Grundlegend werden die drei Web-Crawler-Metriken *Abdeckung*, *Aktualität* und *Durchsatz* verwendet, die bereits in Kapitel 3.1.3 beschrieben wurden. Da die Metriken nicht in der gewünschten Weise zur Verfügung stehen und teilweise auch Vereinfachungen sinnvoll sind, werden für die Erprobung angepasste Metriken verwendet.

Anstelle der Abdeckung wird lediglich die *Anzahl der besuchten Seiten in der URL-Datenbank* gemessen. Die *Anzahl gewünschter Seiten*, welche als Faktor im Nenner der Abdeckungsgleichung (siehe Gleichung 1, Seite 7) steht, kann als Konstante gesehen werden und beeinflusst alle Werte für die Abdeckung gleichermaßen. Die Anzahl der besuchten Seiten wird am Ende eines Testfalles gemessen. Es muss nicht die Differenz zum Startwert gemessen werden, da der Startwert zu Beginn eines Testfalles immer auf den gleichen Wert zurückgesetzt wird. Es wird somit eine Vereinfachung und bessere Lesbarkeit gewählt. Um die Anzahl der besuchten Seiten zu ermitteln, werden die beiden Datenbankstatistiken *Anzahl der URLs* und das *Verhältnis der besuchten URLs* miteinander multipliziert, wie in der folgenden Gleichung dargestellt ist.

Gleichung 4: Besuchte Seiten einer Datenbank

$$Besuchte\ Seiten = db\_url\_amount \cdot db\_visited\_ratio$$

Auch die Aktualität wird in einer angepassten Version gemessen. Einerseits kann nicht ermittelt werden, ob und welche Seiten zum Zeitpunkt der Messung aktuell sind, andererseits ist das Konzept der Aktualität im Simulator-System sehr schwierig abbildbar. In der Realität würde das Metatag `last_modified` des HTTP-Headers darüber Auskunft geben. Das *Alter der Datenbank* wird von Cho und Garcia-Molina (2003, S. 393) als alternative Metrik empfohlen. In dieser Arbeit wird dazu der Mittelwert aller `url_last_visited`-Daten[21] der URLs in der Datenbank gebildet und in den Datenbankstatistiken als `db_avg_freshness` zurückgegeben. Aufgrund der unterschiedlichen Startzeiten der Tests, wird der Startzeitpunkt des jeweiligen Tests vom `db_avg_freshness`-Wert subtrahiert. Wird dies nicht getan, ist das Datenbankalter von älteren Tests automatisch ein niedrigeres als das Datenbankalter von neuen Tests. Es entsteht die neue Metrik *Aktualität'*:

Gleichung 5: Aktualität der Datenbank

$$Aktualität' = db\_avg\_freshness - iter\_fetch\_start$$

Der Durchsatz wird im weiteren Verlauf nicht verwendet. Stattdessen wird die Auslastung (engl. utilization) der Fetcher-Komponenten berechnet. Für den Zeitraum des Simulationsvorgangs stehen die Laufzeit und die Prozessorzeit zur Verfügung. Die Auslastung ist der Anteil der Laufzeit, in dem der Prozessor verwendet wurde. Die Auslastung kann somit mit den Iterationsmesswerten wie folgt berechnet werden:

Gleichung 6: Auslastung einer Fetcher-Komponente

$$Auslastung = \frac{iter\_fetch\_cpu\_time}{iter\_fetch\_time}$$

Mithilfe der Auslastung kann erkannt werden, welche Einstellungskombinationen für die Fetcher-Komponenten geeignet sind. Diese Metrik soll verwendet werden, um die Implementierung der parallelen Prozesse und parallelen Fetcher auf Funktion zu überprüfen.

Die Webseiten-Metriken Wichtigkeit und Änderungsrate werden im Testsystem nicht berechnet. Der Wert für die Wichtigkeit wird per Zufall bei der Generierung einer URL bestimmt. Diese Generierung wird im folgenden Unterkapitel beschrieben. Die Änderungsrate wurde nicht im Testsystem implementiert, kann aber in zukünftigen Arbeiten ergänzt werden.

7.1.5 Generierung von simulierten Werten

Die Werte für den PageRank (Webseiten-Metrik für Wichtigkeit), die Verzögerungsdauer (Crawl Delay) und die Auswahl einer zufälligen Top-Level-Domain werden auf der Grundlage von Verteilungen der Realität bzw. Annäherungen aus der wissenschaftlichen Literatur zufallsgeneriert erstellt.

[21] Hier ist der Plural von Datum gemeint.

Die Generierung eines *PageRank-Wertes* geschieht in zwei Schritten. Im ersten Schritt wird eine Zahl zwischen 0 und 60 Milliarden zufällig generiert. Diese Zahl repräsentiert den Rang in der Liste aller URLs. Im zweiten Schritt erfolgt die Umwandlung dieses Ranges in einen PageRank-Wert. Ein PageRank kann Werte zwischen 0 und 10 annehmen und kann durch das Potenzgesetz beschrieben werden. Die initial generierte Rangzahl wird im zweiten Schritt einem entsprechenden Bereich zugewiesen. Die Zuweisungen der Bereiche wurden bereits in Tabelle 20 aufgelistet. Ein Rang von 1.400 ergibt beispielsweise einen zufälligen PageRank zwischen 1,0 und 2,0. Mithilfe dieser Zuordnung können realitätsnah sehr viele, sehr kleine und sehr wenige, sehr hohe PageRank-Werte erstellt werden. Auf ähnliche Weise wurde eine Verteilung in (Broder et al. 2006, S. 134) generiert.

Tabelle 20: Zuordnung der Webseitenränge zu PageRank-Wertebereichen

Zufälliger Rang		Zufallsbereich für PageRank	
von	**bis**	**min**	**max.**
1	10	8,0	10,0
11	100	4,0	8,0
101	1.000	2,0	4,0
1.001	10.000	1,0	2,0
10.001	100.000	0,2	1,0
100.001	1.000.000	0,01	0,2
1.000.001	10.000.000	0,001	0,01
10.000.001	100.000.000	0,0001	0,001
100.000.001	1.000.000.000	0,00001	0,0001
1.000.000.001	60.000.000.000	0,000001	0,00001

Die Generierung von Werten für die *Verzögerungsdauer* (Crawl Delay) besitzt als Grundlage die Verteilung aus einer bestehenden Untersuchung (siehe Abbildung 23). Der meistangegebene Wert auf Webseiten ist 10 Sekunden. Allerdings sind 80 % der Webseiten nicht in dieser Grafik dargestellt, da diese keinen Wert angegeben haben. Der Mittelwert aller angegebenen Werte ist 47 Sekunden, der Mittelwert von angegebenen Werten unter einer Minute ist 20 Sekunden.

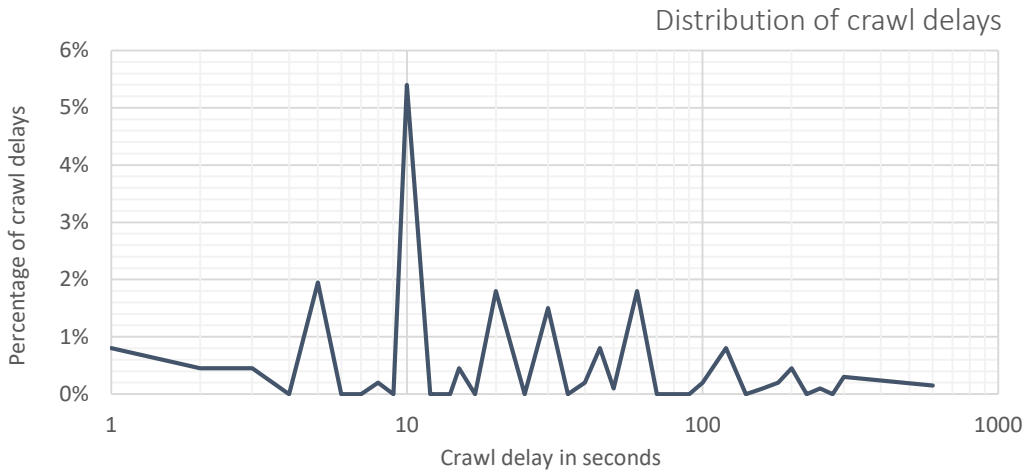

Abbildung 23: Verteilung der Werte für die Verzögerungsdauer, logarithmische Ansicht (angelehnt an Kolay et al. 2008, S. 1172)

Die Generierung von zufälligen *Top Level Domains* entspricht der aktuellen Verteilung der Top-Level-Domains im Internet (Domain Name Stat 2020). Es wurden die 200 Top-Level-Domain-Bezeichnungen mit den meisten Registrierungen verwendet. Sieben Bezeichnungen wurden aufgrund von Kompatibilitätsbedenken verworfen. Dabei handelte es sich um internationalisierte Top-Level-Domain-Bezeichnungen der Art `.pф`. Die Liste und die Anzahlen der verwendeten Top Level Domains kann in *Anhang D* betrachtet werden. Die Generierung der Top Level Domains findet Anwendung bei der Generierung von neuen Domains in der Steuerungskomponente oder im Fetcher-Simulator. Zudem wird eine Top Level Domain mit dem gleichen Generator erzeugt, wenn ein Fetcher erstellt wird. Die Top Level Domain wird bei dem Fetcher als bevorzugte Domain eingetragen und für die Top-Level-Domain-Partitionierung verwendet.

7.1.6 Weitere Anmerkungen zum Testsystem

Die Testeinstellungen sehen vor, dass ein Fetcher eine vorgegebene Anzahl an Iterationen durchlaufen kann. Im Gegensatz dazu ist auch denkbar, die Fetcher-Komponente(n) für eine vorgegebene Zeit laufen zu lassen und die Statistiken davon abzufragen. Diese Alternative wurde nicht implementiert und kann ggf. in zukünftigen Arbeiten bearbeitet werden. Für das Testsystem bedeutet die verwendete Vorgehensweise, dass weniger Kommunikationsaufwand notwendig ist. Die Fetcher-Komponenten laufen autonom und werden lediglich über die AWS-Konsole gestartet und beendet.

Es wurde für alle Tests eine Einstellung im Testsystem deaktiviert, die in einer Produktivumgebung aktiviert sein soll. Bei der Übermittlung der verarbeiteten FQDN-Listen an den Ergebnissammler werden die Reservierungseinträge dieser FQDN-Fetcher-Kombination nicht aus der Datenbank entfernt. Dies hat den Hintergrund, dass die Listen komplett abgearbeitet werden können und wiederholte Abrufe der gleichen Listen verhindert wird. Dies ermöglicht es, die Metrik *besuchte Seiten* nachvollziehbar und wiederholbar für die Einzelfälle messen zu können.

7.2 Durchführung der Testfälle

Es folgt die Darstellung und Durchführung der Testfälle. Zu Beginn wird die Funktionstüchtigkeit der Implementierung auf Plausibilität und Grenzen getestet. Dafür werden verschiedene Anzahlen von Prozessen innerhalb der Fetcher-Komponenten und verschiedene Anzahlen von parallellaufenden Fetcher-Komponenten erprobt. Es folgen Tests bezüglich der Strategien für die Partitionierung und Priorisierung der FQDN-Listen. Dabei werden lediglich Langzeitstrategien erprobt, Kurzzeitstrategien werden nicht gegenübergestellt, da davon ausgegangen wird, dass immer alle URLs einer Domain übertragen werden. Eine Übersicht der zu testenden Fälle ist Tabelle 21 zu finden.

Es folgen die Beschreibungen der einzelnen Testfälle. Die Ergebnisse bauen teilweise aufeinander auf und werden somit direkt in die Beschreibung der Testfälle eingefügt. Die Zusammenfassung, Gegenüberstellung und gewonnene Erkenntnisse folgen in Kapitel 7.3.

Für jeden Test werden zu Beginn die Rahmenbedingungen, die Einstellungen und die bereits bekannten Einschränkungen und gegebenenfalls dahingehende Anpassungen aufgenommen. Es werden in den Testbeschreibungen lediglich die Werte, die sich von den Standardwerten unterscheiden, genannt. Die Standardwerte der Testeinstellungen sind in *Anhang C* aufgelistet, die ausführlichen Testeinstellungen können in *Anhang E* betrachtet werden. Zudem wird beschrieben welche Strategie bzw. Systemtechnik verglichen wird und welche Metrik für das Ergebnis verwendet werden.

Die Ergebnisse werden in Diagrammen dargestellt. Für die Auslastung, den Anteil besuchter Seiten und Aktualität' werden die Log-Daten der Fetcher-Simulatoren ausgewertet. Für die Auslastung der Steuerungskomponente, des Ergebnissammlers und der URL-Datenbank wird der Amazon Service *Cloudwatch* verwendet. In diesem Service können verschiedene Metriken der bereitgestellten Systeme eingesehen werden. Diese Analyse erfolgt rein visuell, da die Daten nicht exportierbar sind und nur in der Weboberfläche von Amazon betrachtet werden können.

Tabelle 21: Übersicht der zu untersuchenden Testfälle

Test	Strategie(n)	Variable(n)	Ergebnismetrik(en)
Partitionierung durch parallele Prozesse	Zufall	`parallel_processes`, `fqdn_amount`	Auslastung pro Fetcher Auslastung der Komponenten im Testsystem
Partitionierung durch parallele Fetcher	Zufall	`parallel_processes`, `parallel_fetcher`	Auslastung der Komponenten im Testsystem
Partitionierung durch URL-Informationen	Top-Level-Domain, FQDN-Hash, Consistent Hashing, Große-Seiten-Zuerst	–	Anteil besuchter Seiten, Aktualität' Auslastung der Komponenten im Testsystem
Priorisierung nach Größe	Große-Seiten-Zuerst, Kleine-Seiten-Zuerst, Zufall	–	Anteil besuchter Seiten
Priorisierung nach Datum	Neue-Seiten-Zuerst, Alte-Seiten-Zuerst, Zufall	–	Aktualität', Auslastung der Komponenten im Testsystem, Anteil besuchter Seiten

Für den Vergleich der Auslastung der Komponenten ist der zeitliche Verlauf und die Auslastung der im Weiteren überwachten Systeme einmal im Leerlauf durchgeführt worden. Den Leerlauf beschreibt Abbildung 24. Die Auslastung im Leerlauf der Steuerungskomponente und des Ergebnissammlers schwankt zwischen 0,3 % und 0,5 %. Die Auslastung der URL-Datenbank liegt in einem Bereich von 3,0 % bis 4,0 %.

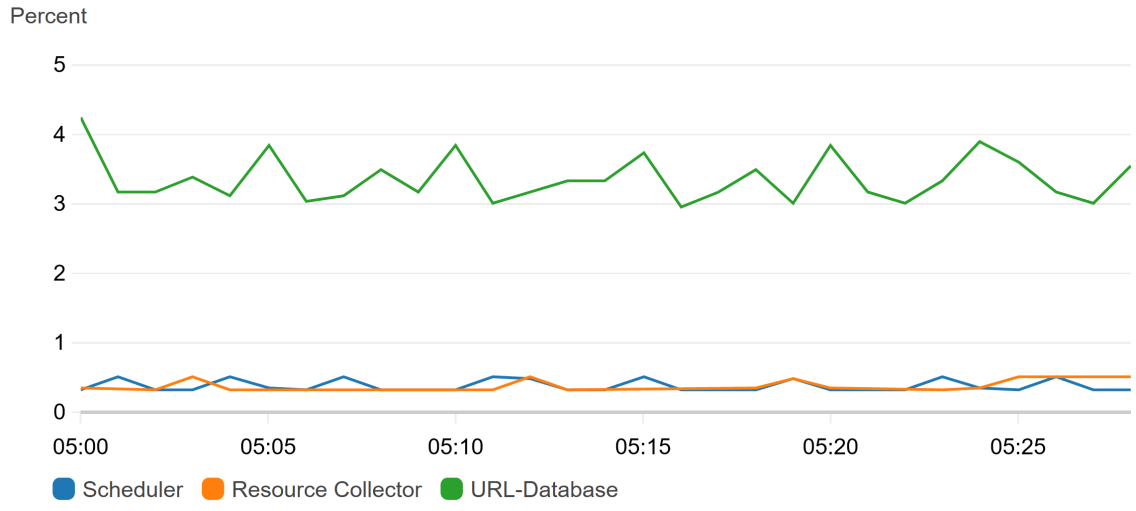

Abbildung 24: Auslastung des Testsystems im Leerlauf

7.2.1 Parallele Prozesse innerhalb einer Fetcher-Komponente

Im ersten Test wird überprüft, ob die lokale Parallelisierung funktioniert und mit welcher Anzahl an Prozessen eine hohe Auslastung und kurze Bearbeitungszeit erreicht werden kann. Innerhalb der Fet-

cher-Simulatoren werden die FQDN-Listen zur Abarbeitung einem Thread-Pool-Manager zugewiesen. Dieser Manager weist jedem Thread genau eine FQDN-Liste zu. Die verfügbare Anzahl der Threads ist durch die Fetcher-Einstellungen `parallel_process` vorgegeben. Die Anzahl der FQDN-Listen wirkt sich auf die Auslastung einer Fetcher-Komponente aus und wird mit in die Erprobung aufgenommen.

Aus den Kombinationen der verschiedenen Anzahlen abzurufender FQDN-Listen und den verschiedenen Anzahlen zu verwendender Prozesse der Fetcher-Komponente entstehen insgesamt 165 Einzeltests. Der Testfall wird durch die Einstellungen in Auszügen in Tabelle 22 beschrieben.

Tabelle 22: Einstellungen für die Erprobung paralleler Prozesse (Auszug)

Kategorie	Einstellung	Wert(e)
Projekteinstellungen	Durchläufe	1
Beispieldatenbank	Anzahl FQDNs	1.000
	Anzahl URLs	5.000 (5 pro FQDN)
Testfall	Abzurufende FQDN-Listen	4 - 44 (Schrittweite 4)
	Parallele Prozesse	2 - 30 (Schrittweite 2)
	Bearbeitungsgeschwindigkeit[22]	10

In jedem Einzeltest werden Dauer und Prozessorzeit für den jeweiligen Fetch-Vorgang gemessen. Mithilfe der Gleichung 6 (siehe Seite 47) wird die Auslastung berechnet. Dauer und Auslastung werden in der folgenden Analyse den variablen Einstellungswerten Anzahl FQDN-Listen und parallele Prozesse gegenübergestellt.

Die Gegenüberstellung der Dauer für die einzelnen Fetch-Vorgänge zeigt, dass die Fetcher-Simulatoren mit vorgegebenen 26 und 30 parallelen Prozessen nicht korrekt funktionieren, dargestellt ist dies in Abbildung 25. Die Zeiträume für diese Fetch-Vorgänge erhöhen sich teilweise auf Werte über 20 Sekunden, obwohl der Durchschnitt aller ermittelten Werte bei 4,4 Sekunden, bereinigt um Ausreißer lediglich 1,1 Sekunden, liegt.

[22] Die Verzögerungsdauer wurde in diesem Fall durch 10 geteilt, um eine zehnfach beschleunigte Abarbeitung zu erzielen.

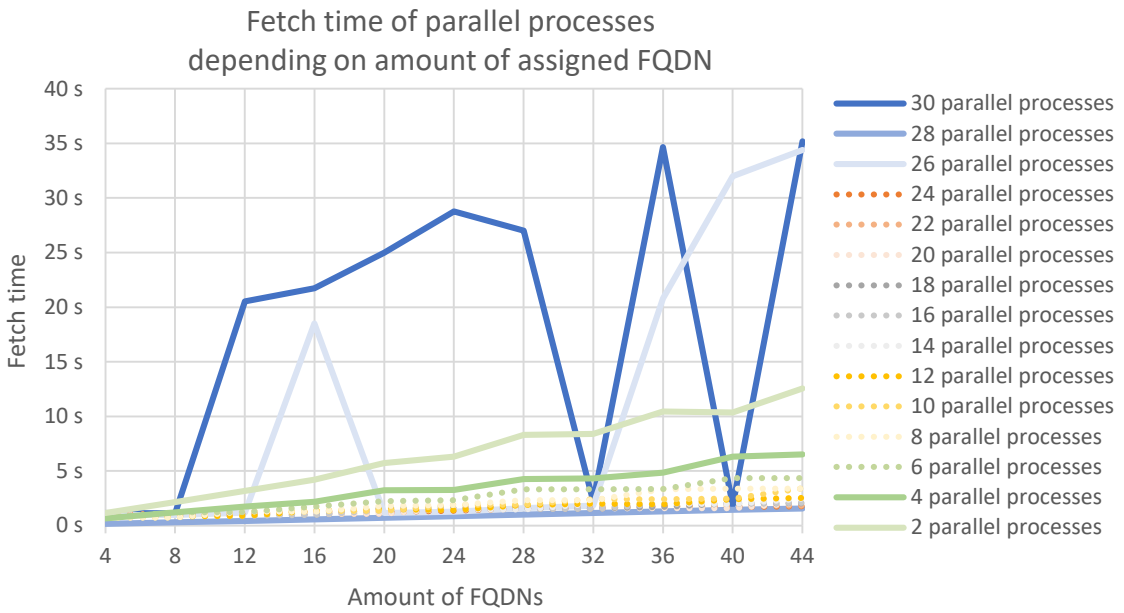

Abbildung 25: Zeiten für Fetch-Vorgänge mit parallelen Prozessen

Die Fetcher-Simulatoren besitzen lediglich einen virtuellen Kern und werden durch die hohe Anzahl an parallelen Prozesse überlastet. Daher werden die drei Wertereihen der Fetcher-Simulatoren mit 26, 28 und 30 parallellaufenden Prozessen im weiteren Verlauf nicht verwendet.

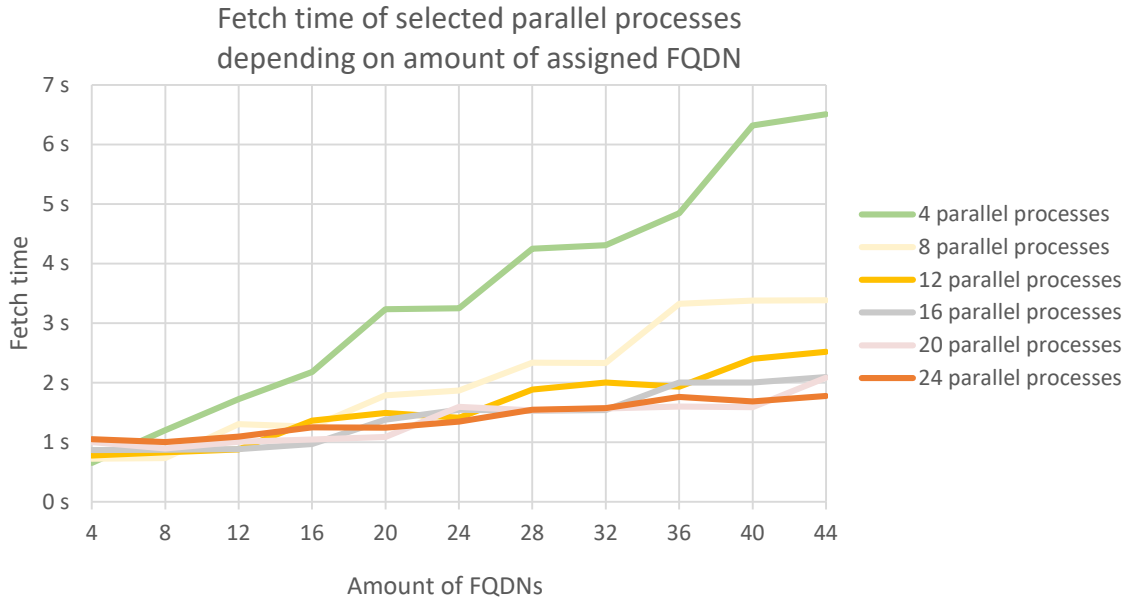

Abbildung 26: Zeiten für Fetch-Vorgänge ausgewählter paralleler Prozesse

Eine auf sechs ausgewählte Wertereihen reduzierte Darstellung der Zeiträume erfolgt in Abbildung 26. Das Diagramm zeigt, dass die Verarbeitung einer höheren FQDN-Anzahl zu einer größeren Dauer des Fetch-Vorgangs führt. Dies gilt für alle Einstellungswerte der Anzahl paralleler Prozesse.

Der Vorgang für die Verarbeitung von vier FQDN-Listen beträgt bei allen Anzahlen paralleler Prozesse ungefähr eine Sekunde. Für die Verarbeitung von 44 FQDN-Listen wird bei einer höheren Anzahl an parallelen Prozessen weniger Zeit (ungefähr zwei Sekunden) benötigt als bei einer niedrigen

Anzahl (ungefähr sechs Sekunden). Der Unterschied der Dauer für die Verarbeitung von wenigen FQDN-Listen gegenüber vielen FQDN-Listen wird kleiner, je mehr parallele Prozesse zur Verfügung stehen.

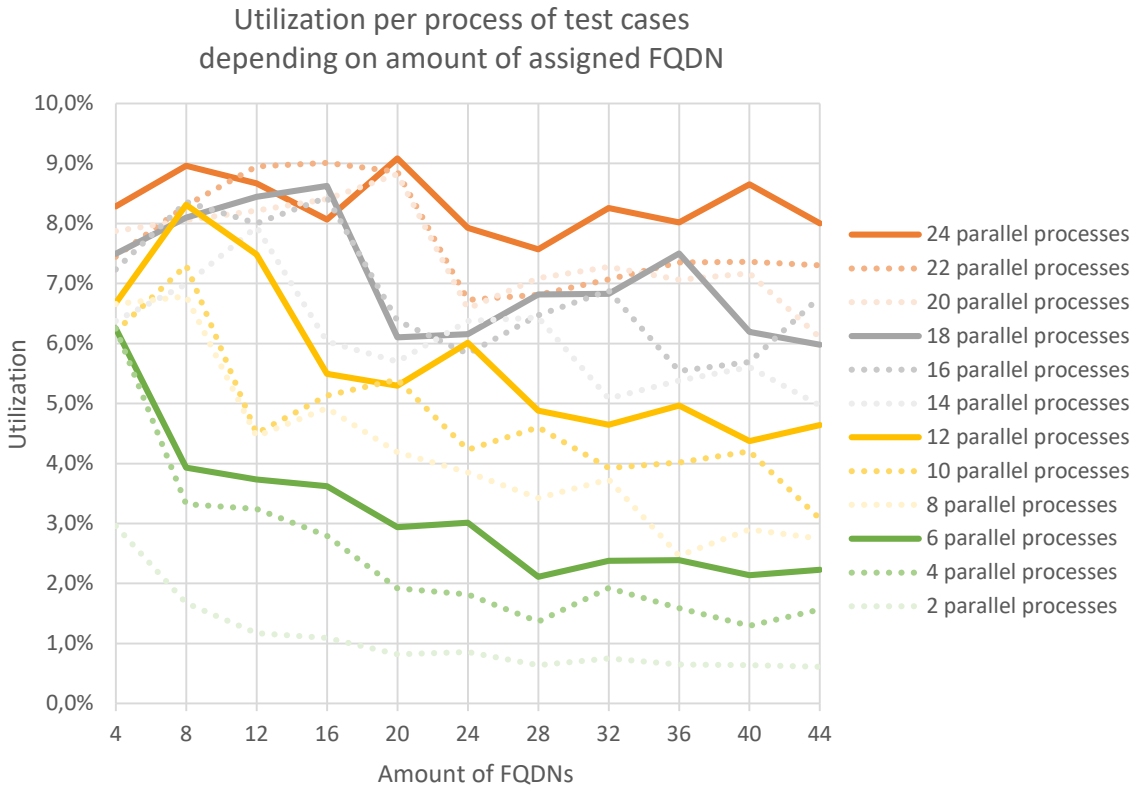

Abbildung 27: Auslastung pro verwendetem Prozess der Testfälle

Die Ermittlung und Gegenüberstellung der Auslastung aller Kombinationen zeigt, dass die Auslastung eines einzelnen Prozesses höher ist, wenn gleichzeitig andere Prozesse im gleichen System laufen. Die höchste Auslastung eines Systems wird immer nahe des Bereiches mit der Anzahl an Prozessen erreicht, die ähnlich der Anzahl der zu bearbeitenden FQDN-Listen ist. Abbildung 27 zeigt außerdem, dass die Prozesse in diesem Test mit höchstens 9 % ausgelastet sind. Eine Erhöhung der Anzahl an zu bearbeitenden FQDN-Listen erhöht die Auslastung nicht. Im Gegensatz dazu, kann die Erhöhung der Anzahl der parallelen Prozesse, falls die verwendete Komponente dies unterstützt, die Auslastung weiter erhöhen.

Die Auslastung der drei Komponenten Steuerungskomponente, Ergebnissammler und URL-Datenbank (engl. Scheduler, Resource Collector, URL-Database) wird in Abbildung 28 aufgeführt. Für die einzelnen Tests mit Anzahlen bis 24 parallelen Prozessen konnte eine eindeutige zeitliche Zuordnung erfolgen. Die Tests mit den parallelen Prozessen der Anzahl 26 bis 28 konnten lediglich teilweise zugeordnet werden, mehrere dieser Tests endeten erst mit starker Verzögerung.

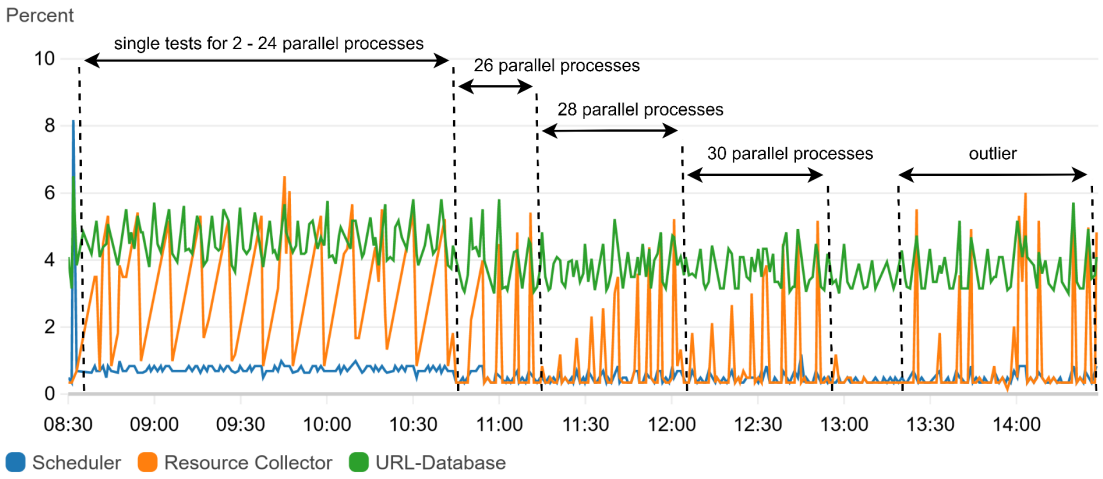

Abbildung 28: Auslastung weiterer Komponenten im Testsystem (zzgl. Zuordnung der Tests der parallelen Prozesse)

Ebenso ist zu erkennen, dass die einzelnen Ausschläge der Testreihe zu Beginn niedriger sind und im weiteren Verlauf ansteigen. Dies resultiert aus der Reihenfolge. Es wurden zu Beginn Tests mit kleinen Anzahlen von FQDN-Listen durchgeführt und schrittweise erhöht. Besonders gut zu erkennen ist dies bei den Tests mit den Werten der 26 bis 30 parallelen Prozesse. Je höher die Anzahl der verarbeiteten FQDN-Listen ist, desto mehr URLs werden (simuliert) gefunden und an den Ergebnissammler zur weiteren Verarbeitung übertragen.

Die Steuerungskomponente ist abgesehen von der initialen Beispieldatenbankgenerierung zu Beginn des Testfalls gering ausgelastet. Der Ergebnissammler und die URL-Datenbank sind mit Maximalwerten von 6 % ebenfalls gering ausgelastet.

7.2.2 Parallele Fetcher-Komponenten innerhalb des Testsystems

Zusätzlich zur lokalen Parallelisierung erfolgt im nächsten Testfall eine verteilte Parallelisierung innerhalb des Testsystems. Die Ergebnisse der einzelnen Fetcher-Komponenten sind nicht Gegenstand der Analyse, da sich diese den Ergebnissen des ersten Testfalls gleichen. Stattdessen werden die Werte der Auslastung der Steuerungskomponente, des Ergebnissammlers und der URL-Datenbank gemessen. Während des Testlaufs gab es teilweise Unterbrechungen, die durch Lücken in den jeweiligen Graphen dargestellt sind.

Tabelle 23: Einstellungen für die Erprobung parallel laufender Fetcher-Komponenten

Kategorie	Einstellung	Wert(e)
Projekteinstellungen	Durchläufe	4
Beispieldatenbank	Anzahl FQDNs	1.000
	Anzahl URLs	5.000 (5 pro FQDN)
Testfall	Parallel Prozesse	8-32 (Schrittweite 8)
	Parallel Fetcher	8-56 (Schrittweite 8)
	Abzurufende FQDNs (pro Fetcher-Komponente)	16
	Iterationen	1

Der Testvorgang besteht aus 28 Kombinationen der Anzahl paralleler Prozesse innerhalb der Fetcher-Komponenten und der Anzahl paralleler Fetcher im Testsystem. Jeder Einzeltest wurde vier Mal wiederholt. Somit wurden insgesamt 112 Testdurchgänge ausgeführt und 1.584 Log-Dateien der erstellten Fetcher-Komponenten zusammengefasst. Tabelle 23 listet die relevanten Testeinstellungen auf.

Abbildung 29 (Seite 57) zeigt die Auslastung der beteiligten Komponenten im Testsystem, in dem die Fetcher-Simulatoren gestartet werden. Dargestellt sind die Steuerungskomponente, der Ergebnissammler und die URL-Datenbank.

Die Abbildung ist aufgeteilt in die vier Auslastungsverläufe der jeweils zur Verfügung gestellten parallelen Prozesse (Prozessanzahl: 8, 16, 24 und 32). Die X-Achse zeigt jeweils den zeitlichen Verlauf. Zudem repräsentiert dieser Verlauf die Anzahl der parallelen Fetcher, welche mit 8 parallelen Fetcher-Komponenten rechts beginnend und mit vier Wiederholungen und der Schrittweite 8 zu einem Wert von 56 parallelen Fetcher-Komponenten auf der linken Seite des jeweiligen Diagramms führt.

Die Auslastung der Steuerungskomponente und der URL-Datenbank steigt mit Erhöhung der Anzahl an Fetcher-Komponenten leicht an. Die Werte der Steuerungskomponente befinden sich zwischen 1,5 % bei wenigen parallelen Fetcher-Komponenten und 11 % bei vielen parallelen Fetcher-Komponenten. Die Werte der URL-Datenbank befinden sich zwischen 5 % und 15 %. Die Anfangswerte sind jeweils 1 % über der jeweiligen Auslastung im Leerlauf. Aus der Änderung der Anzahl paralleler Prozesse resultieren lediglich geringe Änderung von maximal 2 % erhöhter Auslastung.

Die Auslastung des Ergebnissammlers steigt ebenfalls auf den Diagrammen von links nach rechts an. Aufgrund der erhöhten Anzahl an parallelen Fetcher-Komponenten wird die Anzahl der übertragenen FQDN-Listen erhöht. Die Werte der Auslastung liegen zwischen 10 % und 90 % bei Fetcher-Komponenten mit acht gleichzeitig zur Verfügung stehenden Prozessen. Die Auslastung kann durch die Anzahl der parallelen Prozesse innerhalb aller Fetcher-Komponenten noch gesteigert werden. Durch die verringerte Dauer der Verarbeitung von weiteren zur Verfügung stehenden Prozessen wird die gleiche Anzahl an FQDN-Listen in einem kürzeren Zeitraum an den Ergebnissammler zurückgegeben. Dieser bearbeitet die Rückgaben ebenfalls in kürzerer Zeit ab und ist in diesem Zeitraum kurzzeitig zu 100 % ausgelastet.

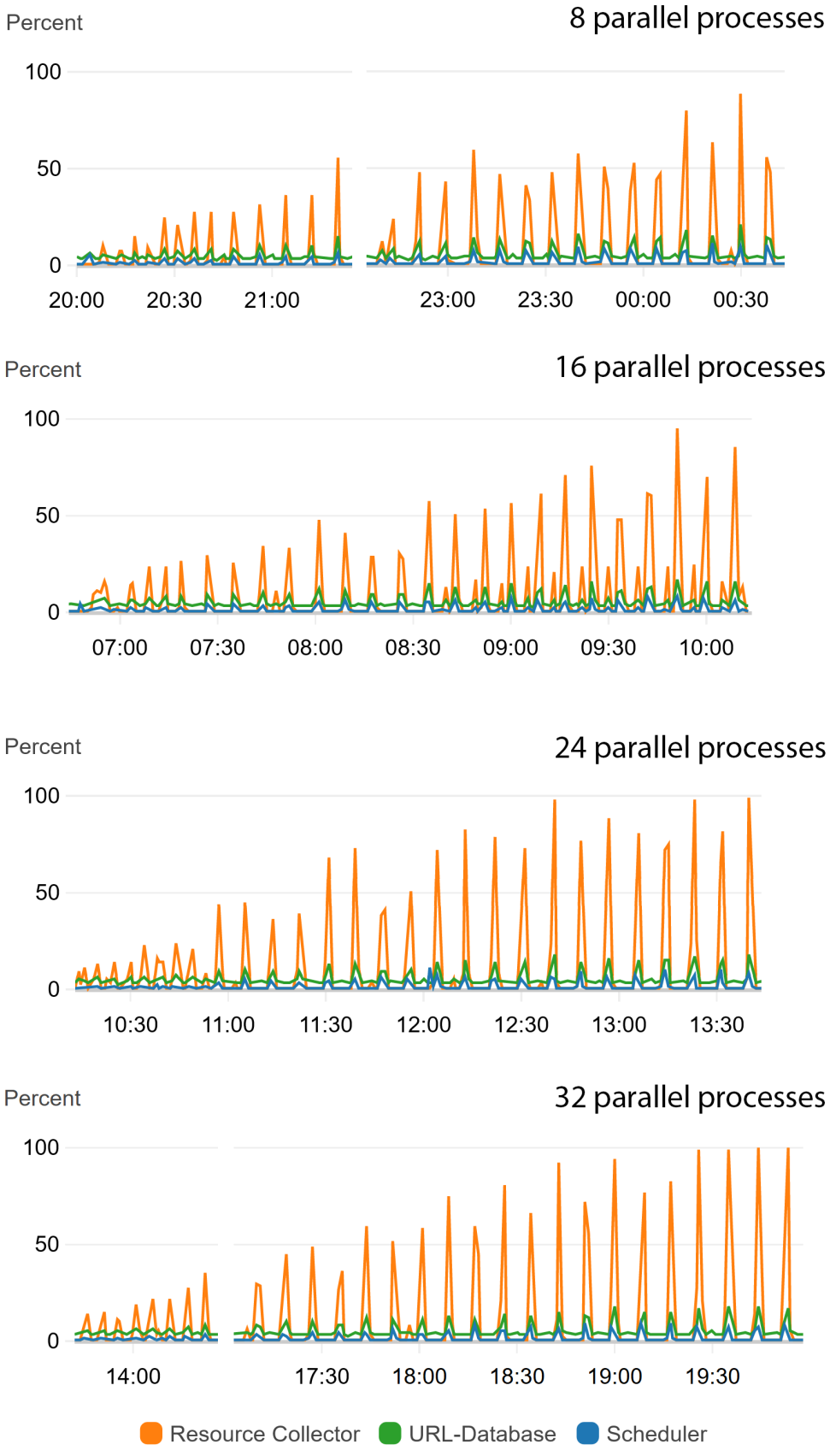

Abbildung 29: Auslastung im Testsystem während der Ausführung paralleler Simulatoren

7.2.3 Partitionierungsstrategien

Durch die Partitionierung der Gesamtliste aller URLs soll gewährleistet werden, dass die Webseiten und deren Infrastruktur nicht überbeansprucht wird. Dies geschieht in der Steuerungskomponente durch die implementierte verschachtelte Warteschlange (siehe Kapitel 4.3.1, Seite 23), wodurch eine Domain genau einem Fetcher zugewiesen wird. Auf der verschachtelten Warteschlange aufbauend, werden in diesem Unterkapitel weitere Partitionierungsstrategien auf Funktion und Metriken überprüft. Das Ziel ist es, die Domains gleichmäßig auf die verfügbaren Fetcher-Komponenten aufzuteilen.

Für den Testfall wird die Anzahl der Fetcher-Simulatoren und deren verfügbare parallele Prozesse auf jeweils 16 fixiert. Zu den drei Partitionierungsstrategien TLD, FQDN-Hash und Consistent Hashing wird ein Test ohne Partitionierungsstrategie durchgeführt. In diesem Fall wird die Priorisierungsstrategie Alte-Seiten-Zuerst angewandt, um eine gleichbleibende Sortierung über die Iterationen hinweg zu gewährleisten. Die Beispieldatenbank besteht aus 4.000 Domains mit jeweils 5 Unterseiten. Die Fetcher-Simulatoren rufen jeweils 50 Domains von der Steuerungskomponente ab, simulieren den Fetch-Vorgang und übermitteln das Resultat an den Ergebnissammler. Der Vorgang wird in fünf Iterationen ausgeführt. In jeder Iteration werden zusammengezählt 800 Domains abgerufen. In fünf Wiederholungen werden somit alle initial generierten 4.000 Domains an die Fetcher-Simulatoren übertragen. Für den Fetch-Vorgang ist eingestellt, dass jede besuchte Seite fünf Links zu bisher unbekannten Unterseiten der aktuell besuchten Domain beinhaltet. Die URL-Datenbank enthält zum Ende des Testdurchlaufs 120.000 URLs, wobei lediglich 20.000 dieser URLs besucht werden können, da die Reservierungsfreigabe für die Testfälle deaktiviert ist. Das optimale Ergebnis des Anteils besuchter Seiten in der URL-Datenbank ist somit 16,67 %. Zusammenfassend sind die Einstellungen in Tabelle 24 aufgelistet.

Tabelle 24: Einstellungen für die Erprobung verschiedener Partitionierungsstrategien

Kategorie	Einstellung	Wert(e)
Projekteinstellungen	Durchläufe	1
Beispieldatenbank	Anzahl FQDNs	4.000
	Anzahl URLs	20.000 (5 pro FQDN)
Testfall	Parallel Prozesse	16
	Parallel Fetcher	16
	Iterationen	5
	Abzurufende FQDNs	50
	Partitionierungsstrategien	ohne, Top Level Domain, FQDN Hashing, Consistent Hashing
	Priorisierungsstrategie	Alte-Seiten-Zuerst
	Links pro Unterseite	5
	Interne Links	100 %
	Unbekannte Links	100 %

Für den Testfall wurde der Anteil der besuchten Seiten, die Aktualität' der Datenbank und die Auslastung der Steuerungskomponente, des Ergebnissammlers und der URL-Datenbank ermittelt.

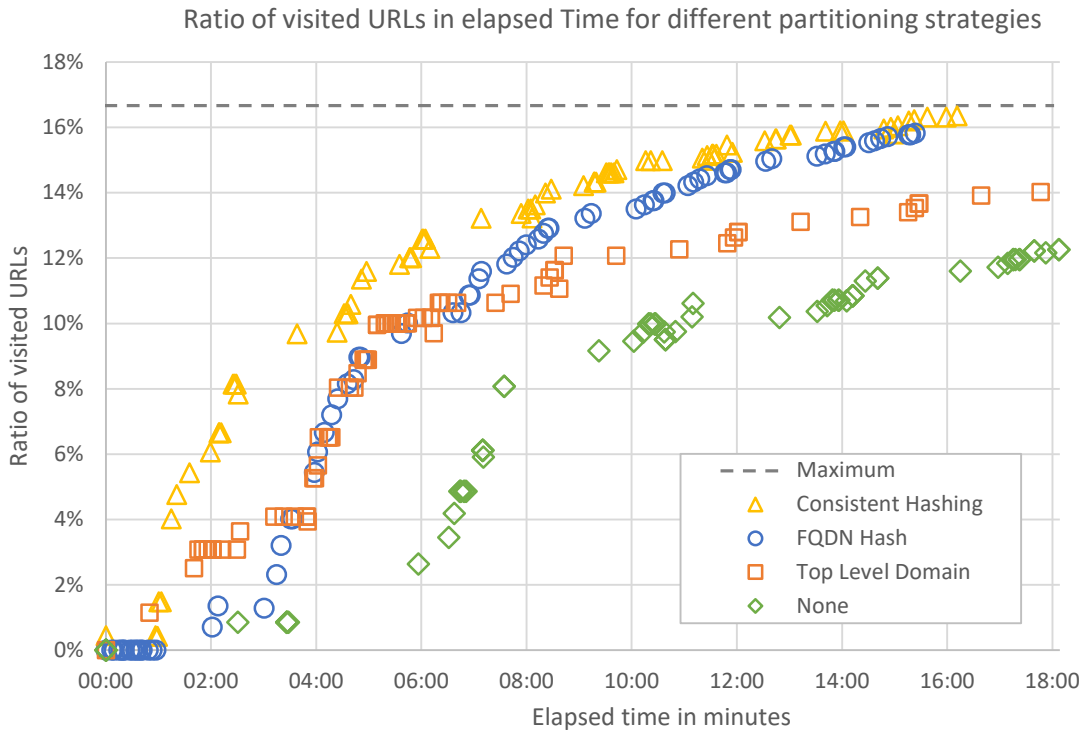

Abbildung 30: Verhältnis besuchter Seiten unterschiedlicher Partitionierungsstrategien

In Abbildung 30 wird der zeitlichen Verlauf des Anteils der besuchten Seiten, abhängig von der verwendeten Strategie, veranschaulicht. Beide hashbasierten Strategien erreichen mit einem Verhältnis besuchter Seiten von ungefähr 16 % nach 16 Minuten nahezu das Maximum. Die Partitionierung

nach Top Level Domain und die Durchführung ohne Partitionierungsstrategie erreichen ihr Maximum jeweils nach 18 Minuten. Der Anteil besuchter Seiten der Top-Level-Domain-Strategie beträgt 14,3 %. Der Testdurchlauf ohne Partitionierungsstrategie erreicht mit einem Anteil von 12,7 % besuchter Seiten den geringsten Wert der Erprobung.

Der geringe Anteil besuchter Seiten mit der Top-Level-Domain-Strategie ergibt sich daraus, dass im Testsystem FQDN-Listen mit einer Top Level Domain existieren können, die keine der Fetcher-Komponenten als bevorzugte Top Level Domain eingetragen hat. Dieses Problem ist in der Realität ebenso gegeben. Der niedrige Anteil des Durchlaufs ohne Priorisierungsstrategie kann lediglich auf einen fehlerhaften Durchlauf zurückgeführt werden. Zu Beginn sollten bereits 16 Fetcher-Komponenten gestartet werden, diese lassen sich im Diagramm und den Logdateien nicht korrekt zuordnen. Die erste Iteration wird in der Datenlage für diesen Fall nur zur Hälfte durchgeführt.

Die Untersuchung der Aktualität' folgt in Abbildung 31. Die gestrichelte Linie beschreibt den Optimalfall, in dem das Alter aller URLs jederzeit null beträgt. Die Werte der Strategien gehen im ersten Drittel des Testfalls auseinander, gleichen sich jedoch zum Ende wieder aneinander an. Da in der Metrik Aktualität' lediglich die Seiten in die Berechnung einfließen, die bereits besucht wurden, entsteht für die Strategien mit geringen Anteilen besuchter Seiten kein Nachteil.

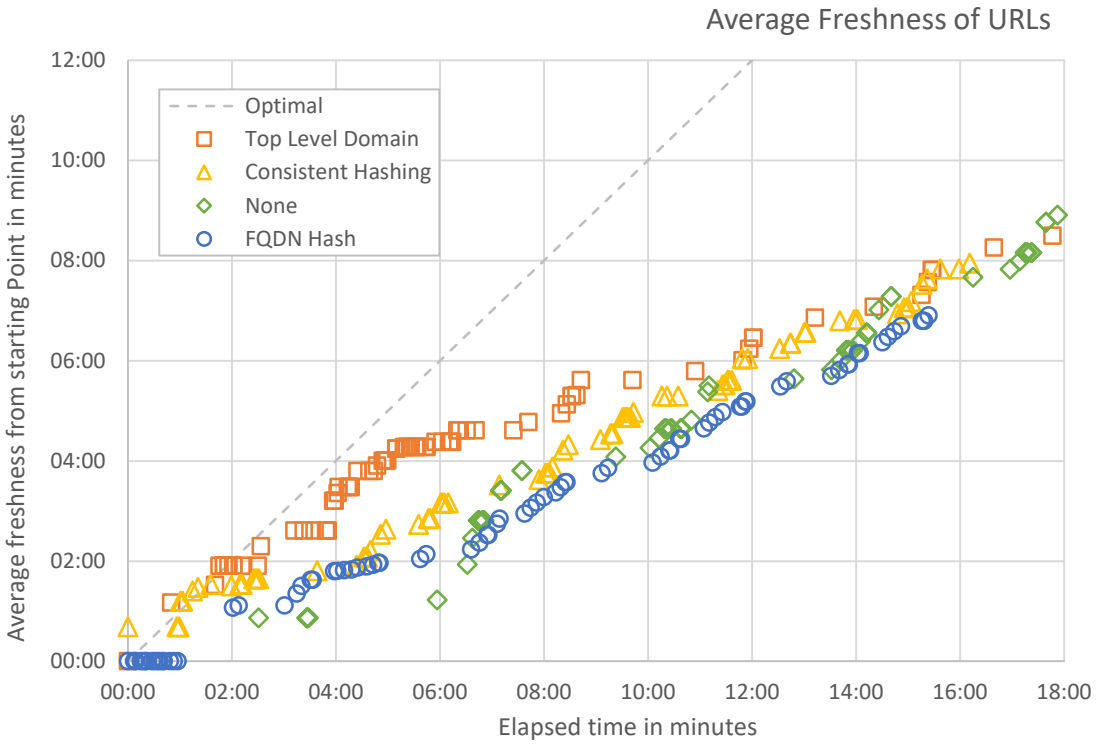

Abbildung 31: Aktualität' verschiedener Partitionierungsstrategien

Den zeitlichen Verlauf der Auslastung im Testsystem zeigt Abbildung 32. Die Strategien wurden in der folgenden Reihenfolge ausgeführt: None, Top Level Domain, FQDN Hash und Consistent Hashing.

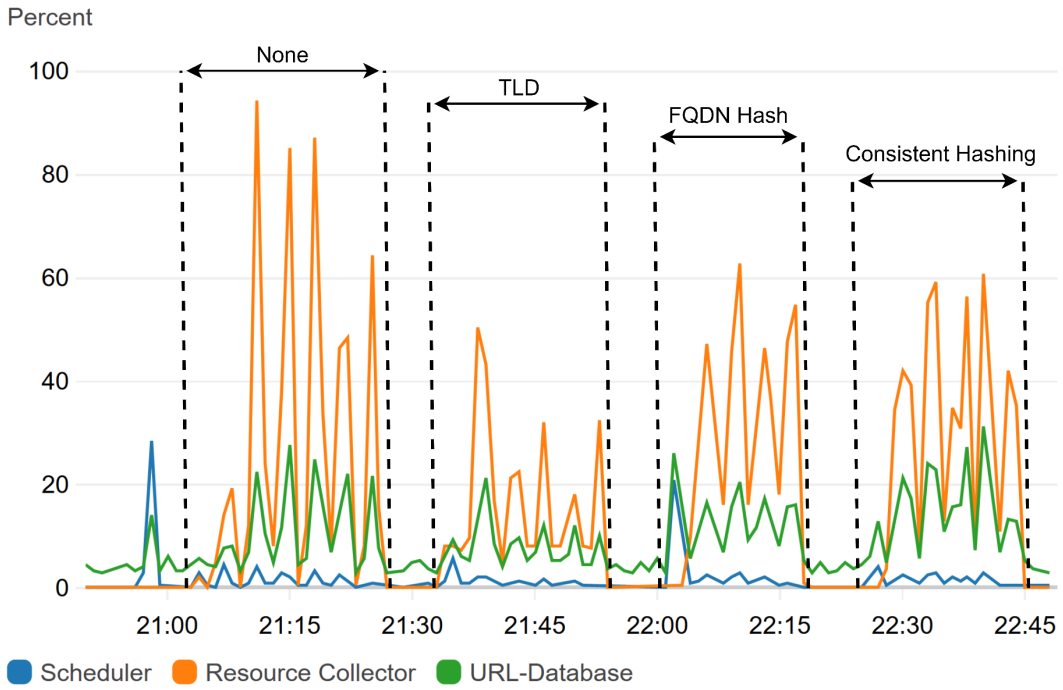

Abbildung 32: Auslastung im Testsystem der Partitionierungsstrategien

Die höchste Auslastung in der Steuerungskomponente und im Ergebnissammler wird im Durchlauf ohne Partitionierungsstrategie erreicht. Dies resultiert aus der erhöhten Abarbeitungsfrequenz, die Ausschläge sind dementsprechend schmaler. Die Auslastung der URL-Datenbank erreicht die höchsten Werte beim Durchlauf ohne Partitionierungsstrategie und bei der Verwendung der Consistent-Hashing-Strategie.

Die Verwendung der Top-Level-Domain-Strategie bewirkt im Testsystem die geringste Auslastung aller Ressourcen, zudem verringert sich die Auslastung während des Durchlaufs. Zu Beginn der FQDN-Hash-Strategie sind die Steuerungskomponente und die URL-Datenbank im Gegensatz zu den anderen Strategien mit mehr als 20 % ausgelastet. Dies begründet sich aus der Neuberechnung der Hashwerte für die Fetcher-Komponenten und der Übermittlung an die Datenbank bei der Initierung der 16 Fetcher-Komponenten.

In einem weiteren Test wird verglichen, inwieweit sich die Auslastung der Komponenten im Testsystem bei Verwendung der Hash-Strategien verhält, wenn sich die Anzahl der Fetcher-Komponenten kontinuierlich verändert. Die Neuberechnung der Hashwerte innerhalb der FQDN-Hash-Strategie ist laut Literatur aufwendig. Die Consistent-Hashing-Strategie benötigt diese Neuberechnung nicht.

Die Testeinstellungen ändern sich für diesen Test nicht. Es wird lediglich mit einem weiteren Programm im Intervall von 15 Sekunden ein neuer Fetcher erstellt, der jedoch keine FQDN-Listen abruft. Den Vergleich der Auslastung im Testsystem zeigt Abbildung 33. Für den Vergleich wurden die Werte am ersten Ergebnissammler-Ausschlag, jeweils in der Abbildung durch einen grauen Balken markiert, ermittelt.

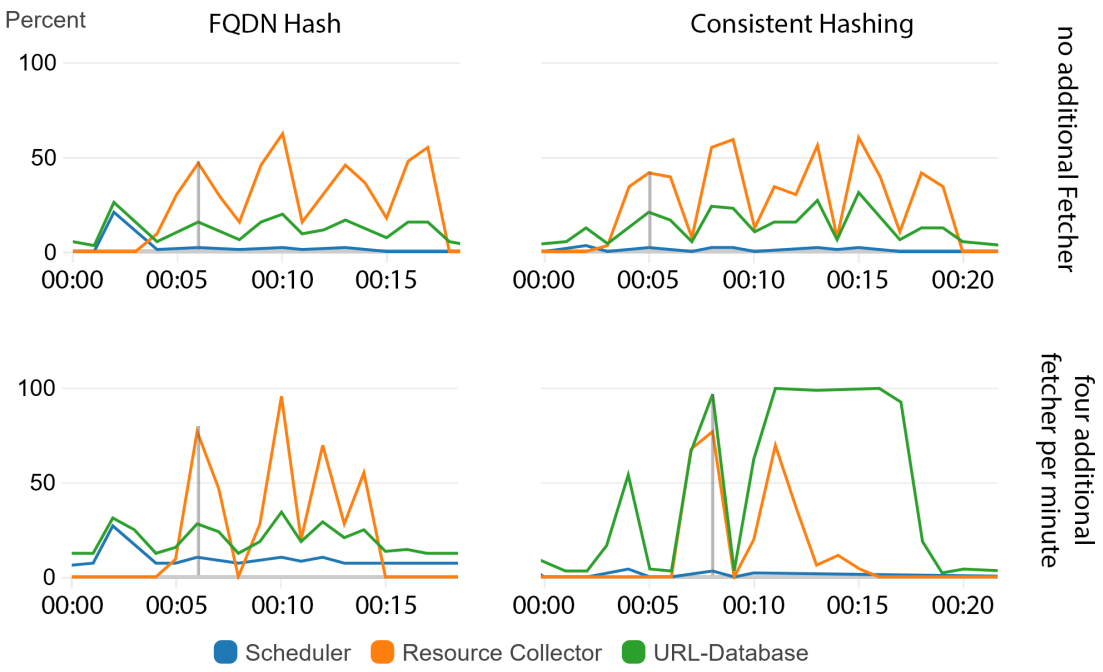

Abbildung 33: Vergleich der Hash-Strategien mit und ohne Änderung der Fetcher-Anzahl

Das Hinzufügen von Fetcher-Komponenten beeinflusst die Auslastung der Testkomponenten bei Verwendung beider Strategien. Für die FQDN-Hash-Strategie gilt, dass die Auslastung für die Startgenerierung der Fetcher-Komponenten innerhalb der Steuerungskomponente und der URL-Datenbank sich geringfügig erhöht. Im weiteren Verlauf erhöht sich die Auslastung der Steuerungskomponente am Messpunkt von 2,5 % auf 11,33 %, des Ergebnissammlers von 47,17 % auf 76,72 % und innerhalb der URL-Datenbank von 16,39 % auf 28,33 %. Das Hinzufügen von Fetcher-Komponenten in regelmäßigen Abständen lastet das Testsystem somit gleichmäßig stärker aus.

Für die Consistent-Hashing-Strategie gilt, dass die URL-Datenbank bei der initialen Generierung der Fetcher-Komponenten und der Domains und URLs statt mit 10 % mit mehr als 50 % ausgelastet wird. Der weitere Verlauf zeigt eine gering ausgelastete Steuerungskomponente, deren Auslastung sich von 2,5 % auf lediglich 4 % erhöht. Der Ergebnissammler ist ohne zusätzliche Fetcher-Generierung mit 42,17 % ausgelastet, mit zusätzlicher Fetcher-Generierung steigt die Auslastung auf 77,46 %, flacht jedoch im weiteren Verlauf ab. Die URL-Datenbank wird statt 21,38 % mit 96,67 % fast vollständig ausgelastet. Es existiert ein folgender zweiter Ausschlag von 100 %, der für 8 Minuten gehalten wird. Durch die regelmäßige Änderung der Fetcher-Anzahl ist die Auslastung im Testsystem ungleichmäßig gestiegen. Die Auslastung der Steuerungskomponente steigt geringer im Vergleich zur FQDN-Hash-Strategie, die Auslastung des Ergebnissammlers steigt in ähnlicher Form. Die URL-Datenbank-Auslastung steigt im Gegensatz zur FQDN-Strategie sehr stark an.

Begründen lässt sich die starke Auslastung der URL-Datenbank durch den gleichzeitig hohen Aufwand der Erstellung von mehreren Hashwerten in einem Fetcher, der Speicherung dieser Werte in einer Verbindungstabelle und der Abfrage für einen Fetcher, der diese Hashwerte für die Filterung der FQDN-Listen verwendet.

7.2.4 Priorisierungsstrategien hinsichtlich der Anzahl der Unterseiten

Der nächste Testfall ist einer von zwei Testfällen, die die Priorisierung von FQDN-Listen betrachten. Es wird in diesem Testfall die Anzahl der Unterseiten als Grundlage für die Priorisierung verwendet, um den Anteil der besuchten Seiten in der URL-Datenbank zu erhöhen. Es werden dafür die Priorisierungsstrategien Große-Seiten-Zuerst und Kleine-Seiten-Zuerst überprüft. Die theoretische Überlegung für diesen Testfall erfolgte bereits in Kapitel 4.2.1 (Seite 19). Den beiden Strategien werden in der Auswertung die zufallsgesteuerte Priorisierungsstrategie gegenübergestellt.

Die Testeinstellungen, dargestellt in Tabelle 25, werden für dieses Szenario angepasst, um eine flexible Generierung neuer Domains und URLs während der Fetch-Vorgänge zu erhalten. Dafür wird ein Anteil an externen Links und ein Anteil an unbekannten Links, die bei der Simulation einer Seitenverarbeitung generiert werden, eingeführt. Bei der Verarbeitung verweisen nun 15 % der Links einer Seite auf externe Unterseiten und 5 % der Links auf bereits bekannte Unterseiten. Beide Einstellungen treten in Kombination auf. Durch die Generierung von bereits bekannten, teilweise auf Domänen liegenden Seiten, die der jeweilige Fetcher nicht kennt, erfolgen vermehrt Abfragen von den Fetcher-Simulatoren an die Steuerungskomponente. Um diese Belastung zu reduzieren, wurde die Zahl der parallellaufenden Fetcher-Komponenten auf vier reduziert.

Tabelle 25: Einstellungen für die Erprobung von Priorisierungsstrategien bezüglich der Seitenanzahl

Kategorie	Einstellung	Wert(e)
Projekteinstellungen	Durchläufe	1
Beispieldatenbank	Anzahl FQDNs	4.000
	Anzahl URLs	20.000 (5 pro FQDN)
Testfall	Parallel Prozesse	16
	Parallel Fetcher	4
	Iterationen	20
	Abzurufende FQDN-Listen	50
	Priorisierungsstrategien	Große-Seiten-Zuerst, Kleine-Seiten-Zuerst, Zufall
	Links pro Unterseite	1
	Interne Links	85 %
	Unbekannte Links	95 %

Die veränderten Testeinstellungen beeinflussen den maximal zu erreichenden Anteil besuchter Unterseiten innerhalb der URL-Datenbank. Der Maximalwert beträgt 57,125 % und kann wie folgt berechnet werden:

Gleichung 7: Berechnung des maximalen Anteil besuchter Unterseiten

$$Max(Anteil\ besuchter\ Seiten)$$

$$= \frac{Anzahl\ URLs + Anzahl\ URLs \cdot (1 - interne\ Links) \cdot unbekannte\ Links}{Anzahl\ URLs + Iterationen \cdot Abzurufende\ FQDN\text{-}Listen \cdot Anzahl\ Fetcher \cdot URLs\ pro\ FQDN}$$

$$= \frac{20.000 + 20.000 \cdot 0{,}15 \cdot 0{,}95}{20.000 + 20 \cdot 50 \cdot 4 \cdot 5} = \frac{22.850}{40.000} = 57{,}125\ \%$$

Da die Einstellungswerte der internen und unbekannten Links lediglich Richtwerte für die Zufallsgenerierung innerhalb der Fetcher-Komponenten sind, können theoretisch auch höhere Werte erreicht werden.

Der Verlauf der Anteile besuchter URLs ist für die drei verwendeten Strategien in Abbildung 34 dargestellt. Die Priorisierungsstrategie Kleine-Seiten-Zuerst erreicht nach 27 Minuten mit 50 % den höchsten Wert für diesen Anteil. Der höchste Wert der zufallsgesteuerten Priorisierung wird nach 42 Minuten und 46 % Anteil besuchter URLs erreicht. Einen ähnlichen Höchstwert erreicht der Durchlauf mit der Große-Seiten-Zuerst-Strategie. Ein Anteil von 44,5 % wird nach 46 Minuten erreicht.

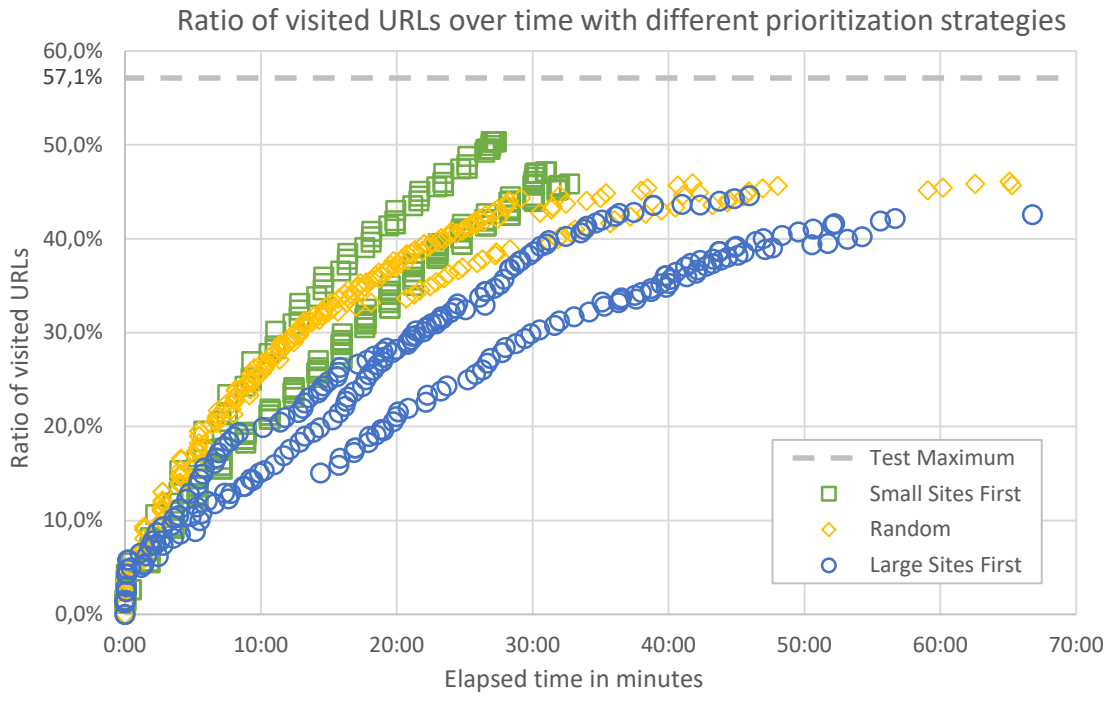

Abbildung 34: Anteil besuchter URLs größenbasierter Priorisierungsstrategien

Das Ergebnis deckt sich nicht mit den theoretischen Überlegungen. Im theoretischen Beispiel werden die Leerlaufzeiten der Große-Seiten-Zuerst-Strategie mit kleinen Seiten aufgefüllt, wodurch eine

schnellere Abarbeitung erreicht wird. Die gegenläufige Kleine-Seiten-Zuerst-Strategie hat im theoretischen Beispiel eine längere Bearbeitungsdauer, da die Leerlaufzeiten der letzten zu bearbeitende Liste nicht durch kürzere Listen aufgefüllt werden kann.

Gegenüber dem theoretischen Beispiel ist im Testfall die Anzahl abzurufender FQDN-Listen vorgegeben. Daraus resultiert, dass die beiden Strategien bei der hohen Anzahl an existierenden Domains, nie die gleichen Listen erhalten. Im theoretischen Beispiel wurde jedoch genau den Fall behandelt, dass die gleiche Liste bearbeitet wurde.

7.2.5 Priorisierungsstrategien hinsichtlich des Alters

Die Durchführung der Testfälle schließt mit der Gegenüberstellung von zwei weiteren Priorisierungsstrategien ab. Es werden die Strategien Alte-Seiten-Zuerst und Neue-Seiten-Zuerst erprobt und der zufallsgesteuerten Priorisierung gegenübergestellt. Das Ziel ist eine hohe Aktualität' innerhalb der URL-Datenbank.

In den Testeinstellungen wurden die Anteile der internen und unbekannten Links jeweils auf 100 % zurückgesetzt. Tabelle 26 zeigt die Übersicht der Testeinstellungen.

Tabelle 26: Einstellungen für Priorisierungsstrategien bezüglich des Alters der FQDN

Kategorie	Einstellung	Wert(e)
Projekteinstellungen	Durchläufe	1
Beispieldatenbank	Anzahl FQDNs	4.000
	Anzahl URLs	20.000 (5 pro FQDN)
Testfall	Parallel Prozesse	10
	Parallel Fetcher	4
	Iterationen	50
	Abzurufende FQDNs	20
	Priorisierungsstrategien	Alte-Seiten-Zuerst, Neue-Seiten-Zuerst, Zufall
	Links pro Unterseite	5
	Interne Links	100 %
	Unbekannte Links	100 %

Den zeitlichen Verlauf der Strategien zeigt Abbildung 35. Die Alte-Seiten-Zuerst-Strategie beendet den Durchlauf nach 13,5 Minuten, die zufallsgesteuerte Priorisierungsstrategie und die Neue-Seiten-Zuerst-Strategie benötigen für die Abarbeitung 18 und 19 Minuten. Die Steigung aller Verläufe ist linear und befindet sich in einem ähnlichen Bereich.

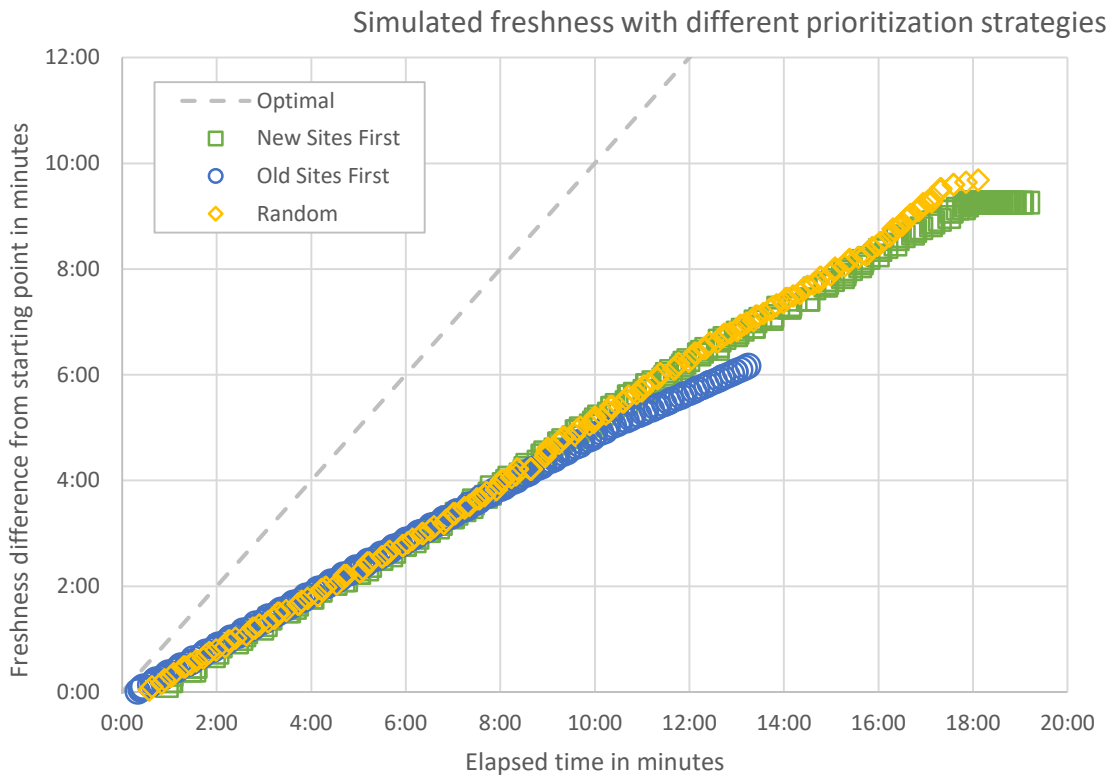

Abbildung 35: Verlauf der Aktualität' unterschiedlicher Priorisierungsstrategien

Die Unterschiede der Auslastung in den weiteren Komponenten des Testsystems wird in der folgenden Abbildung 36 dargestellt. Die Auslastung ist für den Testfall der Alte-Seiten-Zuerst-Strategie höher als die Auslastung der Neue-Seiten-Zuerst-Strategie. Die Auslastung der Strategie mit zufälligen FQDNs liegt zwischen den Werten der beiden anderen Strategien. Die höhere Auslastung des Ergebnisssammlers und der URL-Datenbank bei Verwendung der Alten-Seiten-Zuerst-Strategie zeigt erneut die schnellere Abarbeitung mit dieser Strategie.

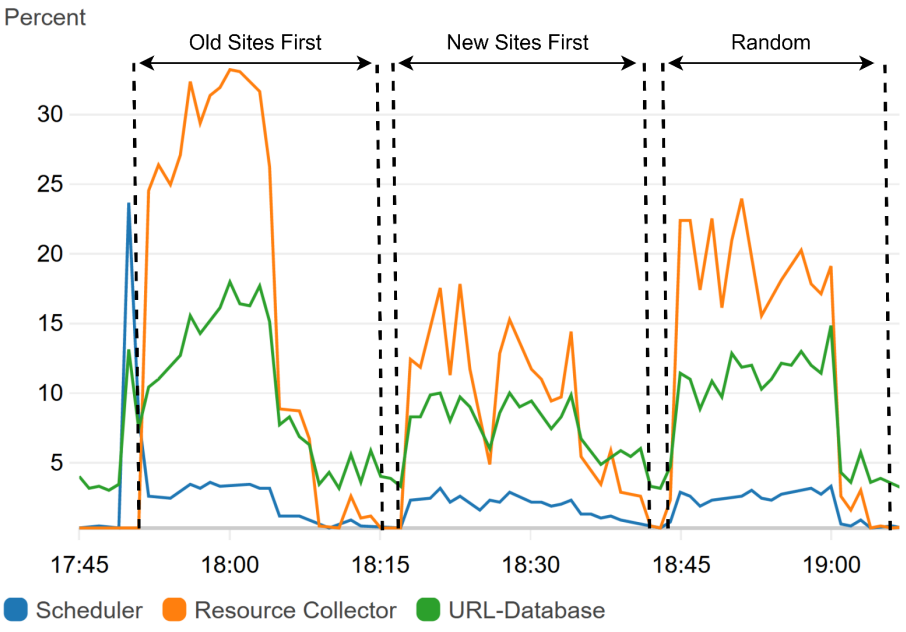

Abbildung 36: Auslastung im Testsystem bei Priorisierungsstrategien hinsichtlich Alter der Seite

Als zusätzliche Gegenüberstellung wurde der Anteil besuchter Seiten für die drei Strategien verglichen (siehe Abbildung 37). Da die Alte-Seiten-Zuerst-Strategie eine schnellere Abarbeitungszeit besitzt wird der Höchstwert in kürzerer Zeit erreicht. Die altersbasierten Strategien erreichen Anteile von 16 %. Die zufallsgesteuerte Strategie erreicht nahezu das Maximum mit 16,6 %.

Der Grund für die schnellere Abarbeitung der Alte-Seiten-Zuerst-Strategie kann mithilfe eines Beispieldurchlaufs erklärt werden. Die Seiten mit den durchschnittlich ältesten Seiten sind zu Beginn immer die Seiten, die noch keine zusätzlichen URLs durch simulierte Fetch-Vorgänge erhalten haben. Somit stehen in diesen Domains weniger Unterseiten für die Simulation zur Verfügung und es wird eine schnellere Abarbeitung ermöglicht.

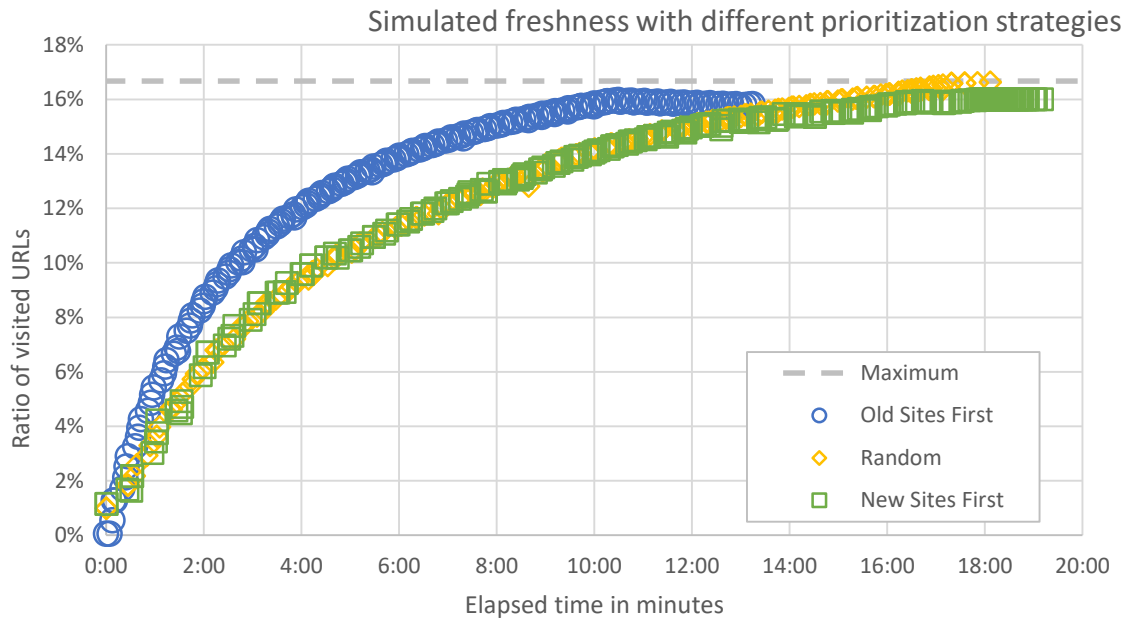

Abbildung 37: Anteil besuchter URLs mit Priorisierungsstrategien hinsichtlich Alter der Seite

7.3 Zusammenfassung der Testergebnisse

Das folgende Unterkapitel stellt die Ergebnisse der Testfälle nochmals in einer Übersicht gegenüber und gibt einen Überblick über die Grenzen der beteiligten Komponenten im Testsystem.

Durch die Testreihen mit parallelen Prozessen im Testsystem wurden folgende Erkenntnisse erarbeitet. Bei maximaler Belastung von 24 parallelen Prozessen und 44 zu verarbeitenden FQDN-Listen wurde eine Auslastung von 9 % gemessen. Durch Erhöhung der verfügbaren parallelen Prozesse kann die Auslastung weiter gesteigert und die Dauer der Bearbeitung verkürzt werden. Die gesteigerte Prozessanzahl sollte allerdings vor der produktiven Verwendung mehrfach überprüft werden. Ein weiteres Ergebnis dieses Testfalls ist, dass die Dauer eines Fetch-Vorgangs verlängert wird, je mehr FQDN-Listen von einer Fetcher-Komponente abgerufen werden sollen. Die Auslastung des Testsystems erhöht sich bei der Verwendung einer Fetcher-Komponente leicht. Die Auslastung der Steuerungskomponente und der URL-Datenbank erhöhen sich gleichbleibend um wenige Prozent, der Ergebnissammler ist je nach Anzahl der übermittelten FQDN-Listen um 1 % bis 6 % höher ausgelastet. Die

Anforderung der Funktion paralleler Prozesse mit Zuweisung der FQDN-Listen innerhalb der Fetcher-Komponenten konnte somit erfüllt werden.

Mit der zusätzlichen verteilten Parallelisierung durch die gleichzeitige Verwendung von Fetcher-Komponenten konnte die Auslastung der Komponenten im Testsystem erhöht werden. Für die Erprobung wurden bis zu 56 parallellaufende Fetcher-Komponenten aktiviert. Für die Steuerungskomponente und die URL-Datenbank entstanden auf diese Weise kurze Ausschläge in ihrer Auslastung von bis zu 15 %. Kurzzeitige Auslastungsausschläge von 100 % im Ergebnissammler wurden mit 56 parallelen Fetcher-Komponenten mit jeweils 32 parallelen Prozessen erzeugt. Der Testfall wurde erfolgreich abgeschlossen. Die Funktionalität der Steuerungskomponente, beliebige viele anfragende Fetcher-Komponenten gleichzeitig mit URLs zu beliefern, ist gegeben.

Im dritten Testfall wurde die Umsetzung der Partitionierungsstrategien Top Level Domain, FQDN Hash und Consistent Hashing auf Funktion überprüft und die ermittelten Ergebnisse verglichen. Alle Strategien konnten erfolgreich ausgeführt werden. Im Testfall konnten mithilfe der Consistent-Hashing-Strategie und der FQDN-Hash-Strategie höhere Werte für den Anteil besuchter Seiten erreicht werden als für die Top-Level-Domain-Strategie. Bezüglich der Aktualität' entwickelten sich alle Strategien zu ähnlichen Werten. In der Gegenüberstellung der beiden hashbasierten Strategien mit kontinuierlicher Änderung der Fetcher-Anzahl wurden für beide Strategien erhöhte Auslastungen gemessen. Im Einzeltest der FQDN-Hash-Strategie erhöhte sich die Auslastung aller Testsystemkomponenten gleichmäßig. Der Einzeltest der Consistent-Hashing-Strategie ergab eine ähnliche Erhöhung der Auslastung für den Ergebnissammler, eine geringere Erhöhung der Auslastung der Steuerungskomponente und eine auf 100 % erhöhte Auslastung der URL-Datenbank.

Die beiden letzten Testfälle beschreiben die Umsetzung der Priorisierungsstrategien Große-Seiten-Zuerst, Kleine-Seiten-Zuerst, Alte-Seiten-Zuerst und Neue-Seiten-Zuerst. Alle Strategien wurden erfolgreich auf ihre Funktion überprüft. Die Gegenüberstellung der Strategien hinsichtlich ihrer Seitengröße ergab einen höheren Anteil besuchter URLs mithilfe der Kleinen-Seiten-Zuerst-Strategie. Die Gegenüberstellung der Strategien hinsichtlich des durchschnittlichen Alters ergab für die Aktualität' keine Unterschiede. Die Alte-Seiten-Zuerst-Strategie simuliert den Fetch-Vorgang der 50 FQDN-Listen in kürzerer Zeit. Durch diese schnellere Bearbeitung ist einerseits die Auslastung der Testsystemkomponenten für den kürzeren Zeitraum erhöht, andererseits kann der Anteil besuchter URLs schneller maximiert werden.

8 Fazit

Abschließend folgt an dieser Stelle das Fazit. Dafür werden die wesentlichen Punkte der Arbeit zusammengefasst und ein Ausblick über weitere Möglichkeiten gegeben, wie die erarbeiteten Erkenntnisse und die realisierten Komponenten von der Open Search Foundation weiterverwendet werden können. Zudem wird auf offene Überlegungen hingewiesen, die in zukünftigen Abschlussarbeiten bearbeitet werden können.

8.1 Zusammenfassung

Das Ziel der Masterarbeit war die Erstellung einer Steuerungskomponente, die in einem verteilten Web-Crawling-System über eine REST-Schnittstelle für Fetcher-Komponenten zur Verfügung stehen soll. Anfragen der Fetcher-Komponenten sollten mit einer URL-Liste beantwortet werden und in der Art und Weise priorisiert und partitioniert sein, dass die Metriken Abdeckung, Aktualität und Durchsatz des Gesamtsystems maximiert werden und gleichzeitig schonend mit Ressourcen außerhalb des Systems umgegangen wird.

Um das Ziel zu konkretisieren wurden zu Beginn der Arbeit die Grundlagen von Web-Crawling-Systemen zusammengefasst. Mit diesem detaillierten Grundverständnis wurden die Architektur und die Standard-Metriken für das zu entwickelnde Web-Crawling-System gewählt. Zudem konnten sich daraus die wichtigsten Anforderungen für das zukünftige System spezifizieren. Diese sind die möglichst gleichmäßige und hohe Auslastung des Gesamtsystems und gleichzeitig die Schonung von fremden Ressourcen (Kapitel 3.2).

Aus den Grundlagen wurden Vorgaben für das zu entwickelnde System erstellt und als Ausgangspunkt für die Analyse der wissenschaftlichen Literatur verwendet (Kapitel 4.1.1). Aus der Analyse entstand eine Gegenüberstellung der Priorisierungs- und Partitionierungsstrategien mit der jeweiligen zu optimierenden Web-Crawler-Metrik. Als optimale Lösung für die nicht messbare Anforderung der Schonung fremder Ressourcen wurde ohne Tests die Partitionierungsstrategie der verschachtelten Warteschlange (Kapitel 4.3.1) ausgewählt.

Im Entwurf wurden die grundlegenden Funktionalitäten der REST-Schnittstelle und der Steuerungskomponente beschrieben. Zusätzlich erfolgte die Verortung der Steuerungskomponente in den Architekturentwurf des Open Web Index. Dieser musste mit weiteren Komponenten, wie der URL-Datenbank versehen werden, um eine lauffähige Steuerungskomponente realisieren zu können (Kapitel 5.3.2). Des Weiteren wurde an dieser Stelle die Auswahl zwingend und optional zu implementierender Strategien durchgeführt. Es folgte die Realisierung der Steuerungskomponenten, der URL-Datenbank und der Priorisierungs- und Partitionierungsstrategien auf Basis des Entwurfskapitels.

Für die Erprobung der implementierten Strategien wurde ein umfangreiches Testsystem erstellt und mit Einzelkomponenten ausgestattet, um einen kompletten Web-Crawling-Kreislauf zu ermöglichen (Kapitel 7.1.1). Für die Steuerung des Testsystems wurde ein separates Programm entwickelt, mit dem die Testfälle automatisiert und wiederholbar durchgeführt werden können. Der Steuerungskomponente wurden zusätzliche Funktionalitäten für die Beispieldatengenerierung zur Verfügung gestellt. Es wurden daraufhin fünf Testfälle erstellt und verfügbare Messwerte identifiziert. Die Standard-Metriken konnten in diesem System nicht ermittelt und weitergegeben werden. Daher entstanden

ähnliche Metriken, die aus den Messwerten verfügbar gemacht wurden. Statt der Metriken *Abdeckung*, *Aktualität* und *Durchsatz* wurden die Metriken *Anteil besuchter Seiten*, *durchschnittliches Alter der Seiten in der URL-Datenbank* und die *Auslastung der Komponenten im System* verwendet, um die Strategien vergleichen zu können (Kapitel 7.1.4).

Die Ergebnisse zeigen, dass die Steuerungskomponente funktioniert und die implementierten Strategien weitestgehend fehlerfrei laufen. Die im Testsystem generierten Fetcher-Komponenten erhalten die Abarbeitungslisten in der vorgegebenen Form der verschachtelten Warteliste. Die Schonung fremder Ressourcen ist somit gewährleistet. Die Implementierung der parallellaufenden Prozesse läuft bis zu Werten von 24 zuverlässig, parallele Fetcher bis zu einer Anzahl von 56 konnten ebenfalls erfolgreich getestet werden (Kapitel 7.2.2). Des Weiteren wurden die zwingend zu implementierenden Partitionierungsstrategien Top Level Domain, FQDN Hash und Consistent Hashing und die Priorisierungsstrategien Zufallsgesteuert, Große-Seiten-Zuerst, Kleine-Seiten-Zuerst, Alte-Seiten-Zuerst, Neue-Seiten-Zuerst und durchschnittlicher PageRank vollständig umgesetzt und abgesehen von der letzten Strategie in einem Testfall erprobt.

Die optional zu implementierenden Strategien wurden nicht entwickelt. Es wurde zudem keine optimale Strategie für die gleichzeitige Maximierung gemessener Metriken gefunden. Hinsichtlich der umfangreichen Literaturanalyse zu Beginn und der aufwendigen Erstellung des Testsystems, um die Erprobungen nachvollziehbar, wiederholbar und automatisiert durchführen zu können, konnten lediglich initiale Tests durchgeführt werden.

8.2 Ausblick

Bisher ist die Steuerungskomponente, so wie das gesamte Testsystem in ressourcensparender Weise angelegt. Trotz der geringen Ressourcen sind weitestgehend keine Grenzen aufgetreten. Falls dies in zukünftigen Arbeiten der Fall sein sollte, stehen zwei Skalierungsmöglichkeiten offen. Die Steuerungskomponente und die URL-Datenbank können entweder mit einer höheren Anzahl an Ressourcen ausgestattet werden oder sie können verteilt betrieben werden.

Vielversprechende, in zukünftigen Abschlussarbeiten zu untersuchende, Strategien sind die geobasierten Partitionierungsstrategien (Kapitel 4.3.3) und die Partitionierung mithilfe von Graph-Datenbanken, die in dieser Arbeit jedoch nicht thematisiert wurden. Die ebenfalls nicht realisierte Priorisierungsstrategie der historischen Änderungsrate (Kapitel 4.2.2) verspricht weitere interessante Ergebnisse.

Zudem ist es möglich, die Optimierung nach anderen Metriken, als nur Abdeckung, Aktualität und Auslastung zu erproben. Insbesondere der im implementierten System nicht realisierte Durchsatz scheint vielversprechend.

9 Quellenverzeichnis

Abiteboul, S., Preda, M., & Cobena, G. (2003). Adaptive on-line page importance computation. In G. Hencsey (Hrsg.), *Proceedings of the 12th international conference on World Wide Web* (S. 280). New York, NY: ACM.

Baeza-Yates, R., Castillo, C., Marin, M., & Rodriguez, A. (2005). Crawling a Country: Better Strategies than Breadth-First for Web Page Ordering. In A. Ellis (Hrsg.), *Special interest tracks and posters of the 14th international conference on World Wide Web* (S. 864–872). New York, NY: ACM.

Bayer, M. (2020). *SQLAlchemy. The Database Toolkit for Python*. https://www.sqlalchemy.org. Zugegriffen: 13. Juni 2020.

Beck, K. (2015). *A Kent Beck signature book: Test-driven development. By example* (20. printing). Boston: Addison-Wesley.

Boldi, P., Codenotti, B., Santini, M., & Vigna, S. (2004). UbiCrawler: a scalable fully distributed Web crawler. *Software: Practice and Experience, 34*(8), 711–726.

Brin, S., & Page, L. (1998). The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems, 30*(1-7), 107–117.

Broder, A. Z., Lempel, R., Maghoul, F., & Pedersen, J. (2006). Efficient PageRank approximation via graph aggregation. *Information Retrieval, 9*(2), 123–138.

Cambazoglu, B. B., & Baeza-Yates, R. (2015). Scalability Challenges in Web Search Engines. *Synthesis Lectures on Information Concepts, Retrieval, and Services, 7*(6), 1–138.

Cambazoglu, B. B., Junqueira, F., Plachouras, V., & Telloli, L. (2008). On the Feasibility of Geographically Distributed Web Crawling. In R. Lempel, R. Perego, & F. Silvestri (Hrsg.), *Proceedings of the Third International ICST Conference on Scalable Information Systems* (S. 31–40): ICST.

Cho, J., & Garcia-Molina, H. (2002). Parallel crawlers. In D. Lassner, D. de Roure, & A. Iyengar (Hrsg.), *Proceedings of the eleventh international conference on World Wide Web - WWW '02* (S. 124). New York, New York, USA: ACM Press.

Cho, J., & Garcia-Molina, H. (2003). Effective page refresh policies for Web crawlers, *28*(4), 390–426.

Cui, Y., Sparkman, C., Lee, H.-T., & Loguinov, D. (2018). Unsupervised Domain Ranking in Large-Scale Web Crawls. *ACM Transactions on the Web, 12*(4), 1–29.

Domain Name Stat (2020). *Domain name registrations in All TLDs*. https://domainnamestat.com/statistics/tldtype/all. Zugegriffen: 20. Juni 2020.

Du, Y. (2020). *xxhash 1.4.4. xxhash is a Python binding for the xxHash library by Yann Collet.* https://pypi.org/project/xxhash. Zugegriffen: 5. Juli 2020.

Edwards, J., McCurley, K., & Tomlin, J. (2001). An adaptive model for optimizing performance of an incremental web crawler. In V. Y. Shen (Hrsg.), *Proceedings of the 10th international conference on World Wide Web* (S. 106–113). New York, NY: ACM.

Encode OSS (2020). *Starlette. The little ASGI framework that shines.* https://www.starlette.io/. Zugegriffen: 13. Juni 2020.

Exposto, J., Macedo, J., Pina, A., Alves, A., & Rufino, J. (2005). Geographical partition for distributed web crawling. In C. Jones (Hrsg.), *Proceedings of the 2005 workshop on Geographic information retrieval* (S. 55–60). New York, NY: ACM.

Exposto, J., Macedo, J., Pina, A., Alves, A., & Rufino, J. (2008). Efficient Partitioning Strategies for Distributed Web Crawling. In T. Vazão, M. M. Freire, & I. Chong (Hrsg.), *Lecture Notes in Computer Science: Information Networking. Towards Ubiquitous Networking and Services* (S. 544–553). Berlin, Heidelberg: Springer Nature.

Huffaker, B., Fomenkov, M., Plummer, D. J., Moore, D., & Claffy, K. Distance Metrics in the Internet .

Huss, N., Lewandowski, D., Sander-Beuermann, W., & Ude, A. (2017). *Proposal for an Open Web Index*. Hannover. https://openwebindex.eu/wp-content/uploads/2019/01/Open_Web_Index_proposal.pdf. Zugegriffen: 9. August 2019.

The Internet Society (1987). Domain Names - Implementation and Specification: The Internet Society, 1987. https://tools.ietf.org/html/rfc1035. Zugegriffen: 18. Juli 2020.

The Internet Society (2005). A Universally Unique IDentifier (UUID) URN Namespace(RFC 4122): The Internet Society, 2005. https://tools.ietf.org/html/rfc4122. Zugegriffen: 18. Juli 2020.

Karger, D., Sherman, A., Berkheimer, A., Bogstad, B., Dhanidina, R., Iwamoto, K., Kim, B., Matkins, L., & Yerushalmi, Y. (1999). Web caching with consistent hashing. *Computer Networks, 31*(11-16), 1203–1213.

Kitchenham, B. (2007). *Guidelines for performing Systematic Literature Reviews in Software Engineering*.

Kolay, S., D'Alberto, P., Dasdan, A., & Bhattacharjee, A. (2008). A Larger Scale Study of Robots.txt. In *WWW 2008* (S. 1171–1172).

Krekel, H. (2020). *pytest. helps you write better programs*. https://docs.pytest.org. Zugegriffen: 13. Juni 2020.

Kumar, M., Bhatia, R., & Rattan, D. (2017). A survey of Web crawlers for information retrieval. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 7*(6), e1218.

Kunder, M. de (2019). *The size of the World Wide Web (The Internet)*. https://www.worldwidewebsize.com/. Zugegriffen: 2. November 2019.

Lee, H.-T., Leonard, D., Wang, X., & Loguinov, D. (2008). *IRLbot*.

Malet, B. (2009). *Distributed Web Crawling Using Network Coordinates*. London.

Nasri, M., & Sharifi, M. (2009). Load Balancing Using Consistent Hashing: A Real Challenge for Large Scale Distributed Web Crawlers. In I. Awan (Hrsg.), *International Conference on Advanced Information Networking and Applications Workshops, 2009. WAINA '09 ; 26 - 29 May 2009, Bradford, United Kingdom ; proceedings* (S. 715–720). Piscataway, NJ: IEEE.

Olston, C., & Najork, M. (2010). Web Crawling. *Foundations and Trends® in Information Retrieval, 4*(3), 175–246.

Quoc, D. L., Fetzer, C., Felber, P., Riviere, E., Schiavoni, V., & Sutra, P. (2015). UniCrawl: A Practical Geographically Distributed Web Crawler. In C. Pu (Hrsg.), *2015 IEEE 8th International Conference on Cloud Computing (CLOUD). June 27, 2015 - July 2, 2015, New York City, New York, USA* (S. 389–396). Piscataway, NJ: IEEE.

Ramírez, S. (2020). *FastAPI. FastAPI framework, high performance, easy to learn, fast to code, ready for production*. https://fastapi.tiangolo.com. Zugegriffen: 13. Juni 2020.

Saini, C., & Arora, V. (2016). Information retrieval in web crawling: A survey. In *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)* (S. 2635–2643): IEEE.

Samuel Colvin (2020). *pydantic. Data parsing and validation using Python type hints*. https://pydantic-docs.helpmanual.io/. Zugegriffen: 13. Juni 2020.

Ueda, T. (2013): *QueueLinker: A Framework for Parallel Distributed Data-Stream Processing*. Phd. Thesis.

Varrazzo, D. (2020). *Psycopg 2. PostgreSQL database adapter for Python*. https://www.psycopg.org/. Zugegriffen: 13. Juni 2020.

ANHANG A Systematic-Literature-Review-Protokoll

Bedarf

Verteilte Crawler sollen von einer Steuerungskomponente, die durch eine REST API angefragt werden kann, URL-Listen zur Abarbeitung erhalten. Die Abarbeitungsliste soll von der Steuerungskomponente in eine geeignete Reihenfolge gebracht werden und unter den verfügbaren Crawlern gleichmäßig aufgeteilt werden.

Forschungsfragen

F1: Welche Arten der Priorisierung von Crawler-Warteschlangen gibt es?
F2: Inwieweit können diese auf verteilte Crawler angewandt werden?
F3: Welche Scheduling-Algorithmen berücksichtigen die Standortvorteile der einzelnen Crawler?
F4: Wie erfolgt ein Vergleich oder eine Evaluation der Techniken?

Strategie

Auswahl der Datenbanken

Es wurden für die Recherche die Datenbanken innerhalb der folgenden Tabelle (Anhang Tabelle 1) verwendet. Die Datenbanken wurden im Oktober 2019 durchsucht.

Anhang Tabelle 1: Verwendete Datenbanken

Datenbank	URL
ACM Digital Library	https://dl.acm.org/
IEEE Xplore Digital Library	https://ieeexplore.ieee.org/
CiteSeerX Library	http://citeseerx.ist.psu.edu/
Google Scholar	https://scholar.google.de/

Weitere nicht verwendete Datenbanken mit Begründung sind in folgender Tabelle (Anhang Tabelle 2) beschrieben.

Anhang Tabelle 2: Nicht verwendete Datenbanken

Datenbank	URL	Begründung für Nicht-Verwendung
Elsevier Science Direct	https://www.sciencedirect.com/	Lediglich 8 Boolean Operatoren pro Abfrage
Springer Link	https://link.springer.com/	Sehr viele Ergebnisse, aber keine Filterung oder Sortierung nach Anzahl der Zitationen möglich

Suchanfragen für die Forschungsfragen

Im Folgenden sind die Suchanfragen aufgelistet, die für die Forschungsfragen erstellt wurden.

F1: Priorisierung von Crawler-Warteschlangen

```
    (crawl OR spider)
AND (frontier OR queue)
AND (priority OR order OR policy OR optimize)
```

F2: Warteschlangen von verteilten Crawler

```
    (crawler OR crawling OR spider)
AND (frontier OR queue)
AND (distributed OR parallel OR 'large scale')
```

F3: Algorithmen mit Berücksichtigung der Standortvorteile beim Scheduling für verteilte Systeme

```
    (scheduling OR scheduler)
AND distributed
AND algorithm
AND geographic
```

F4: Vergleich oder Evaluation von Priorisierungstechniken für Warteschlangen bei verteilten Crawler (Untermenge von F1 und F2)

```
    (crawler OR crawling OR spider)
AND (frontier OR queue)
AND (distributed OR parallel OR 'large scale')
AND (evaluation OR performance OR measure OR comparison OR compare OR
metric)
AND (priority OR order OR policy OR optimize)
```

Auswahlkriterien

Es wurden Artikel ausgewählt und in die eigene Literaturliste übernommen, wenn folgenden Kriterien zutrafen:

- Der Artikel ist in Englisch oder Deutsch verfasst.
- Der Artikel behandelt eine der Forschungsfragen.
- Es werden Artikel aus Journals, von Konferenzen und Sammelbänden aufgenommen. Außerdem wurden für Grundlagen zusätzlich Monographien, Masterarbeiten und Dissertationen aufgenommen.

Auswahlprozedur

Die Suchanfragen ergaben im ersten Schritt 819 Ergebnisse, deren Zuordnung in folgender Tabelle (Anhang Tabelle 3) detailliert dargestellt ist.

Anhang Tabelle 3: Suchergebnisanzahl der einzelnen Anfragen

Plattform \ Forschungsfrage	F1	F2	F3	F4	Summe
ACM Digital Library	9	12	-	5	26
IEEE Xplore Digital Library	12	14	2	4	32
CiteSeerX Library[23]	13	25	12	9	59
Google Scholar[24,25]	218	382	8	94	702
Summe	252	433	22	112	819

Durch die Sichtung der Titel und deren Zusammenfassungen (Abstracts) wurde eine Auswahl der Publikationen erstellt, dargestellt in der nächsten Tabelle (Anhang Tabelle 4). Die Anzahl der ausgewählten Publikationen wurde dadurch auf 122 reduziert.

Anhang Tabelle 4: Anzahl relevanter Ergebnisse der Suche

Plattform \ Forschungsfrage	F1	F2	F3	F4	Summe
ACM Digital Library	7	8	-	4	19
IEEE Xplore Digital Library	10	9	1	4	24
CiteSeerX Library	7	13	5	6	31
Google Scholar[26]	17	22	-	9	48
Summe	41	52	6	23	122

Durch einen weiteren Schritt wurden die Duplikate entfernt, die bei mehreren Suchen gefunden wurden. Das Resultat beinhaltet jede Publikation lediglich ein Mal. Die Anzahl der Publikationen wurde auf 74 reduziert (siehe Anhang Tabelle 5).

Anhang Tabelle 5: Entfernung von Duplikaten

Forschungsfrage	F1	F2	F3	F4
Summe	41	52	6	23
Einzigartige Publikationen		74		

[23] Suchergebnisse gefiltert auf Publikationen mit fünf oder mehr Zitierungen (ohne Filterung: F1:321 / F2:684 / F3: 457 / F4: 261).

[24] Klammerung in Suchabfrage funktionierte hier nicht, daher Suche nach allen Keywords ohne boolesche Operatoren.

[25] Einstellungen: *exclude patents, exclude citations.*

[26] Auswahl lediglich nach Titel.

Im Anschluss (siehe Anhang Tabelle 6) erfolgte eine Klassifizierung der Publikationen. Dabei wurden Publikationen mit nicht passendem Titel oder nicht passendem Abstract herausgefiltert. Publikationen konnten dabei mehreren Forschungsfragen zugeordnet werden.

Anhang Tabelle 6: Kategorisieren der wissenschaftlichen Arbeiten

Forschungsfrage	F1	F2	F3	F4	GL*	WF*	X*
Summe vorher			74				
Summe pro Kategorie[27]	17	20	2	19	13	3	16
Summe einzigartig			53			19	

* GL: Grundlagen WF: weiterführende Forschung X: aussortiert

Die resultierenden 53 wissenschaftlichen Arbeiten wurden im Anschluss genauer geprüft und in einer Skala von 1-5[28] die Relevanz zur jeweiligen Forschungsfrage bewertet. Die Informationen zur Ermittlung der Relevanz wurden einerseits durch die Suche nach relevanten Keywords im Text und andererseits durch die Betrachtung von Tabellen, Grafiken und dem Fazit gewonnen. Dabei wurden teilweise Zugehörigkeiten zu Forschungsfragen / Grundlagen geändert. Durch diese Prüfung verdichtete sich die Auswahl an wissenschaftlichen Arbeiten auf die Anzahlen in folgender Tabelle (Anhang Tabelle 7).

Anhang Tabelle 7: Anzahl der Publikationen mit hoher oder sehr hoher Relevanz

Forschungsfrage	F1	F2	F3	F4	GL*
Relevanz 5 (sehr hoch)	2	1	1	0	3
Relevanz 4 (hoch)	6	11	1	12	7
Summe			34		10

* GL: Grundlagen

In Anhang Tabelle 8 sind alle wissenschaftlichen Arbeiten mit hoher oder sehr hoher Relevanz zu mindestens einer Forschungsfrage oder als Grundlagenwerk eingeschätzt aufgelistet. Die Summe 36 der Arbeiten begründet sich aus der Einschätzung von sieben Arbeiten als relevant für mehrere Forschungsfragen oder einer Forschungsfrage und zugleich als Grundlagenwerk.

[27] Zuweisung von Arbeiten zu mehreren Forschungsfragen möglich, WF und X wurden exklusiv kategorisiert

[28] 1: nicht relevant bis 5: sehr relevant.

Anhang Tabelle 8: Auflistung der Publikationen mit sehr hoher oder hoher Relevanz

Nr.	Autoren	Titel	Jahr	F1	F2	F3	F4	GL
1	Cui, Y., et al.	Unsupervised Domain Ranking in Large-Scale Web Crawls	2018		3		4	
2	Baker, M. R., Akcayol, M. A.	Priority Queue Based Estimation of Importance of Web Pages for Web Crawlers	2017	5				
3	Zhang, L., et al.	DGWC: Distributed and generic web crawler for online information extraction	2016		4		3	
4	Cambazoglu, B. B., Baeza-Yates, R.	Scalability Challenges in Web Search Engines	2015					5
5	Feitelson, D. G.	Workload Modeling for Computer Systems Performance Evaluation	2015					4
6	Le Quoc, D., et al.	UniCrawl: A Practical Geographically Distributed Web Crawler	2015			5	4	
7	Tran, G., et al.	A Random Walk Model for Optimization of Search Impact in Web Frontier Ranking	2015	4			4	
8	Mirtaheri, S. M., et. al	A Brief History of Web Crawlers	2013					4
9	Gupta, S., Bhatia, K. K.	CrawlPart: Creating Crawl Partitions in Parallel Crawlers	2013		4			
10	Yadav, D., et al.	An Approach to design incremental parallel Webcrawler	2012		4			
11	Zheng, S.	Effective methods for web crawling and web information extraction	2011					4
12	Liu, B.; Menczer, F.	Web Crawling	2011					4
13	Ahmadi-Abkenari, F., Selamat, A.	A clickstream-based web page significance ranking metric for Web crawlers	2011				4	
14	Olston, C., Najork, M.	Web Crawling	2010					5
15	Georgios, K.	Distributed Intelligent Crawlers	2010		4		4	
16	Selamat, A., Ahmadi-Abkenari, F.	Application of clickstream analysis as Web page importance metric in parallel crawlers	2010		4		4	
17	Zhou, B., et al.	A distributed vertical crawler using crawling-period based strategy	2010		4		2	
18	Mittal, P., et al.	A scalable, extensible web crawler based on P2P overlay networks	2010		4		1	
19	Soon, L.-K., Lee, S. H.	Reducing Redundant Web Crawling Using URL Signatures	2010	4				
20	Weise, T.	Global optimization algorithms-theory and application	2009					4
21	Manning, C. D., et al.	Introduction to Information Retrieval	2008					5
22	Lee, H.-T., et al.	IRLbot	2008		4		4	
23	Marin, M., et al.	High-performance priority queues for parallel crawlers	2008		5		4	
24	Cambazoglu, B. B., et al.	On the Feasibility of Geographically Distributed Web Crawling	2008			4		
25	Tripathy, A., Patra, P. K.	A Web Mining Architectural Model of Distributed Crawler for Internet Searches Using PageRank Algorithm	2008		4			
26	Guan, Z., et al.	Guide focused crawler efficiently and effectively using on-line topical importance estimation	2008	4				
27	Castillo, C., et al.	A Memory-Efficient Strategy for Exploring the Web	2006	4				
28	Baeza-Yates, R., et al.	Crawling a Country: Better Strategies than Breadth-First for Web Page Ordering	2005	5			4	
29	Srinivasan, P., et al.	A General Evaluation Framework for Topical Crawlers	2005				4	
30	Pant, G., et al.	Crawling the Web	2004					4
31	Castillo, C.	Effective web crawling	2004					4
32	Menczer, F., et al.	Topical web crawlers: Evaluating Adaptive Algorithms	2004	4			4	
33	Boldi, P., et al.	UbiCrawler: A scalable fully distributed Web crawler	2004		4			
34	Ganesh, S., et al.	Ontology-based Web crawler	2004	4				
35	Cho, J.	Crawling the Web: Discovery and Maintenance of Large-Scale Web Data	2001		3		4	
36	Shkapenyuk, V., Suel, T.	Design and implementation of a high-performance distributed Web crawler	2001		4			

Snowballing

Bei der Prüfung der ermittelten Publikationen wurde weitere Literatur durch Verweise im Literaturverzeichnis auf ältere Arbeiten gefunden (*Backward Snowballing).* Zudem wurde mithilfe der Plattform semanticscolar.org Literatur, die auf die bestehende Liste verweist, gesucht und mithilfe einer Analyse des Titels und der Zusammenfassung weitere relevante Werke gefunden.

Die durch die Snowballing-Techniken um 22 Publikationen erweiterte Literaturliste ist in Anhang Tabelle 9 dargestellt.

Anhang Tabelle 9: Weitere mithilfe von Snowballing erfasste Publikationen

Jahr	Autoren	Titel
2019	Chen, J., et al.	Revisiting Consistent Hashing with Bounded Loads
2015	Ahmed, S.T., et al.	Around the web in six weeks: Documenting a large-scale crawl
2013	Takanori Ueda	QueueLinker: A Framework for Parallel Distributed Data-Stream Processing
2011	Cambazoglu, B. B., Baeza-Yates, R.	Scalability Challenges in Web Search Engines
2009	Fetterly, D., et al.	The impact of crawl policy on web search effectiveness
2009	Nasri, M., Sharifi, M.	Load Balancing Using Consistent Hashing: A Real Challenge for Large Scale Distributed Web Crawlers
2008	Exposto, J., et al.	Efficient Partitioning Strategies for Distributed Web Crawling
2008	Olston, C. & Pandey, S.	Recrawl scheduling based on information longevity
2007	Baeza-Yates, R., et al.	Challenges on Distributed Web Retrieval
2006	Castillo, C., et al.	Controlling the Queue Size in Web Crawling
2005	Exposto, J., et al.	Geographical partition for distributed web crawling
2004	Baeza-Yates, R., Castillo, C.	Crawling the Infinite Web: Five Levels Are Enough
2004	Castillo, C., et al.	Scheduling algorithms for web crawling
2003	Cho, J., Garcia-Molina, H.	Effective page refresh policies for Web crawlers
2003	Broder, A.Z., et al.	Efficient URL caching for world wide web crawling
2003	Abiteboul, S. et al.	Adaptive on-line page importance computation
2002	Baeza-Yates, R. & Castillo, C.	Balancing Volume, Quality and Freshness in Web Crawling
2002	Huffaker, B., et al.	Distance Metrics in the Internet
2002	Cho, J., Garcia-Molina, H.	Parallel crawlers
2001	Shkapenyuk, V., Suel, T.	Design and implementation of a high-performance distributed Web crawler
1999	Heydon, A., Najork, M.	Mercator: A scalable, extensible Web crawler
1998	Cho, J., et al.	Efficient crawling through URL ordering

ANHANG B REST-Schnittstelle

Anhang Tabelle 10: REST-Schnittstelle[29] Kategorie Profil

Pfad	Befehl	Beschreibung	Anfrage Werte	Rückgabe Werte
/fetchers/	POST	Neues Fetcher-Konto erstellen	`Kontakt E-Mail*, Fetcher-Name*, Standort, bevorzugte TLD, IP-Adresse`	`UUID, Fetcher Hash(es), Kontakt E-Mail, Fetcher-Name, Erstellungsdatum, Standort, bevorzugte TLD,`
	PUT**	Fetcher-Konto zurücksetzen**	`UUID*, Kontakt E-Mail, Fetcher-Name, Standort, bevorzugte TLD, IP-Adresse`	
	PATCH***	Fetcher-Konto aktualisieren***	`UUID*`	
	DELETE	Fetcher-Konto löschen		

*: zwingend erforderlich **: leere Parameter werden zurückgesetzt ***: leere Parameter werden nicht geändert

Anhang Tabelle 11: REST-Schnittstelle Kategorie URL-Liste

Pfad	Befehl	Beschreibung	Anfrage Werte	Rückgabe Werte
/frontiers/	POST	URL-Liste Abrufen und reservieren	`UUID*, FQDN-Anzahl, URL-Anzahl pro Domain`	`UUID, Rücksende-URL, Ablaufzeitpunkt, FQDN-Anzahl, Gesamt-URL-Anzahl, Liste mit FQDNs, Liste mit URLs`

*: zwingend erforderlich

[29] Online-Version verfügbar unter: http://ec2-18-185-96-23.eu-central-1.compute.amazonaws.com/docs, abgerufen am 19.07.2020.

Anhang Tabelle 12: REST-Schnittstelle Kategorie Entwickerwerkzeuge

Pfad	Befehl	Beschreibung	Anfrage Werte	Rückgabe Werte
/fetchers/	GET	Alle Fetcher auflisten	-	Liste mit Fetcher-Komponenten
/database/	POST	Generieren einer Beispieldatenbank	Fetcher Anzahl, FQDN Anzahl, Anzahl URLs pro FQDN, Anteil besuchter URLs, Festes Crawl Delay	-
	DELETE	komplette oder teilweise Löschung der Beispieldatenbank	Fetcher Hashes, Fetchers, URLs, FQDNs, FQDN-Reservierungen	
/urls/random/	GET	Zufallsgenerierte URL(s) aus Datenbank zurückgeben	Anzahl gewünschter URLs, FQDN	Liste mit URLs
/stats/	GET	Statistiken der Datenbank ausgeben	-	Anzahl Fetcher, Anzahl FQDNs, Anzahl URLs, Anzahl reservierter URLs, Durchschnittliches Alter der URLs, Anteil besuchter URLs
/settings/	GET	Fetcher-Einstellungen abrufen		siehe PUT /settings/
	PUT**	Einzelne Werte in den Einstellungen ändern	Logging Modus, Crawling Speed Faktor, Standardwert für fehlende Crawl Delays, Anzahl paralleler Prozesse, Anzahl paralleler Fetcher, Anzahl Abrufwiederholung, Anzahl abzurufender FQDNs, Anzahl abzurufender URLs pro FQDN, Kurzzeitstrategie Priorisierung, Langzeitstrategie Priorisierung, Langzeitstrategie Partitionierung, Bereich (min, max) zu simulierender URLs pro bearbeitete URL Anteil interne/externe Links in URLs Anteil neu/existierende Links in URLs	

**: leere Parameter werden zurückgesetzt

ANHANG C Einstellungen für Testvorgänge

Anhang Tabelle 13: Testprojekteinstellungen

Eigenschaft	Beschreibung	Standardwert (& Datentyp)
`name`	Aussagekräftiger Name für den Test	- (String)
`repetition`	Anzahl der Wiederholungen des Tests	- (Integer)

Anhang Tabelle 14: Einstellungen der Fetcher-Simulatoren

Eigenschaft	Beschreibung	Standardwert (& Datentyp)
`crawling_speed_factor`	Faktor zur Beschleunigung der Testvorgänge, die Abrufverzögerung wird durch diesen Faktor geteilt	10.0 (Float)
`default_crawl_delay`	Abrufverzögerung, die verwendet werden soll, wenn keine Abrufverzögerung für eine Domain verfügbar ist	10 (Integer)
`parallel_process`	Anzahl der im Fetcher ausgeführten parallelen Prozesse	4 (Integer)
`parallel_fetcher`	Anzahl der parallel zu startenden Fetcher-Instanzen	1 (Integer)
`iterations`	Anzahl der Wiederholungen einer Fetchers-Instanz	1 (Integer)
`fqdn_amount`	Anzahl der pro Iteration maximal abzurufenden Domain-Listen, 0 für alle verfügbaren	30 (Integer)
`url_amount`	Anzahl der pro Domain-Liste maximal abzurufenden URLs, 0 für alle verfügbaren	0 (Integer)
`long_term_prio_mode`	Strategie für die Sortierung der Domain-Listen	*large_sites_first* (String)
`long_term_part_mode`	Strategie für die Aufteilung der Domain-Listen	None (String)
`short_term_prio_mode`	Strategie für die Sortierung der URLs	*new_pages_first* (String)
`min_links_per_page`	Minimale Anzahl an neuen URLs, die im Fetcher-Simulator pro besuchter URL generiert werden	1 (Integer)
`max_links_per_page`	Maximale Anzahl an neuen URLs, die im Fetcher-Simulator pro besuchter URL generiert werden	1 (Integer)
`lpp_distribution_type`	Art der Verteilung der Anzahl der Links einer Seite (links-per-page-distribution)	*discrete* (String)
`int_vs_ext_threshold`	Prozentualer Anteil der simuliert gefundenen URLs, die auf Seiten der eigenen Domain referenzieren. (0.0 = keine, 1.0 = alle)	1.0 (Float)
`new_vs_exist_threshold`	Prozentualer Anteil der simuliert gefundenen URLs, die auf nicht bekannte URLs referenzieren. (0.0 = keine, 1.0 = alle)	1.0 (Float)

Anhang Tabelle 15: Einstellungen der Beispieldatenbank

Eigenschaft	Beschreibung	Standardwert (& Datentyp)
`fqdn_amount`	Menge der Domains	20 (Integer)
`min_url_amount`	Minimale Anzahl an URLs pro Domain	10 (Integer)
`max_url_amount`	Maximale Anzahl an URLs pro Domain	100 (Integer)
`fixed_crawl_delay`	Abrufverzögerung (Crawl Delay) für alle Seiten mit konkretem Wert überschreiben. Die Abrufverzögerung wird bei Standardeinstellung (None) mit simulierten Werten befüllt	None (Integer)
`visited_ratio`	Prozentuale Anzahl der URLs, die als bereits besucht markiert werden	0.0 (Float)

ANHANG D Grundlagen für die Datengenerierung

Anhang Tabelle 16: Verwendete Top Level Domains[30] und deren Verteilung (Zusammengestellt aus Domain Name Stat 2020)

TLD	Anzahl
com	167.447.411
tk	19.226.618
cn	18.661.712
net	16.476.395
de	13.381.471
org	12.854.168
uk	10.611.334
ru	7.573.083
info	7.395.183
top	6.954.036
xyz	4.399.220
ga	4.296.074
fr	4.201.338
cf	4.080.619
icu	4.051.348
nl	3.825.525
eu	3.750.257
ml	3.676.305
loan	3.546.875
ca	3.064.065
us	3.015.729
gq	2.929.550
au	2.817.306
biz	2.814.094
co	2.495.239
it	2.463.202
tw	2.361.056
pl	2.307.803
club	2.278.205
br	2.277.959
in	2.255.275
site	2.198.354
online	1.943.608
es	1.762.619
wang	1.605.728
za	1.520.567
se	1.480.301
vip	1.376.161
cz	1.263.038
jp	1.254.532
be	1.169.228
me	1.162.341
at	1.150.530
dk	1.077.082
cc	1.024.874
ch	997.642
win	877.038
shop	867.954
kr	796.350

TLD	Anzahl
work	780.881
app	757.770
live	739.331
ir	735.794
mx	696.464
pw	670.567
no	633.993
space	598.991
mobi	581.015
nz	562.821
buzz	562.070
io	560.935
ro	550.347
men	549.623
website	531.752
hu	522.615
ua	520.295
fun	506.033
tv	493.967
cl	476.696
bid	467.854
store	459.810
tech	452.274
pro	449.098
fi	444.674
ar	407.679
ltd	405.275
nu	387.549
asia	381.983
gdn	355.205
sk	347.672
tr	334.515
vn	330.403
id	324.853
link	319.019
ws	304.353
life	300.301
gr	299.803
blog	279.051
stream	266.780
ie	264.107
xin	252.537
cloud	244.593
tokyo	243.532
il	241.626
trade	208.851
hk	204.638
kiwi	201.540
review	195.979

TLD	Anzahl
dev	195.547
party	186.866
kz	183.154
ae	181.484
my	175.666
date	172.803
world	167.921
su	163.307
pt	153.962
lt	149.225
host	146.563
ph	143.344
download	141.053
ng	138.589
design	136.528
name	133.120
email	131.840
today	131.217
click	128.998
rocks	126.355
accountant	124.386
sg	120.276
fit	117.965
cat	117.167
ooo	112.673
tel	107.701
solutions	104.682
ke	103.327
science	100.059
one	99.837
xxx	99.406
ee	99.309
group	98.513
bar	97.343
by	96.062
realtor	95.991
press	95.471
racing	93.994
company	91.800
agency	89.878
fm	89.006
london	87.834
best	87.698
nyc	86.917
ovh	86.771
news	85.773
lv	83.555
guru	83.012
la	82.885

TLD	Anzahl
page	82.469
rs	81.504
art	80.099
vg	80.042
monster	79.966
webcam	78.408
faith	76.316
services	73.989
studio	71.131
network	70.335
business	69.794
digital	67.205
pe	66.261
global	65.468
media	63.866
photography	61.902
pub	60.710
hr	60.223
red	59.539
ai	59.467
berlin	57.307
realty	55.830
pk	55.113
center	54.457
uz	54.397
ren	54.010
city	52.983
lu	52.281
th	51.519
ve	50.503
ink	48.678
jobs	47.365
ninja	46.295
academy	45.472
rest	45.048
expert	43.427
technology	43.225
uy	41.667
is	40.912
zone	39.111
systems	37.775
love	37.353
tips	37.002
team	36.658
bg	35.955
video	35.813

[30] Die Liste wurde um internationalisierte länderspezifische Top-Level-Domains wie .pф bzw. xn--p1ai in ASCII-Codierung für Russland aus der Liste entfernt.

ANHANG E Testfälle

Parallele Prozesse

```
example_db_settings = dict(
    fqdn_amount=1000,
    min_url_amount=5,
    max_url_amount=5,
    fixed_crawl_delay=1,
)

project_settings = dict(
    name="parallel_processes",
    date=datetime.now().strftime("%Y-%m-%d"),
    repetition=1,
)

case_settings = pyd.CaseSettings(
    logging_mode=[10],
    crawling_speed_factor=[10.0],
    default_crawl_delay=[1],
    parallel_process=[i*2 for i in range(1,16)],
    parallel_fetcher=[1],
    iterations=[1],
    fqdn_amount=[i*4 for i in range(1,12)],
    url_amount=[0],
    long_term_part_mode=[enum.LONGPART.none],
    long_term_prio_mode=[enum.LONGPRIO.old_sites_first],
    short_term_prio_mode=[enum.SHORTPRIO.old_pages_first],
    min_links_per_page=[3],
    max_links_per_page=[3],
    lpp_distribution_type=[enum.PAGELINKDISTR.discrete],
    internal_vs_external_threshold=[1.0],
    new_vs_existing_threshold=[1.0],
)
```

Parallele Fetcher

```
example_db_settings = dict(
    fqdn_amount=1000,
    min_url_amount=5,
    max_url_amount=5,
    fixed_crawl_delay=1,
)

project_settings = dict(
    name="parallel_fetchers",
    date=datetime.now().strftime("%Y-%m-%d"),
    repetition=4,
)

case_settings = pyd.CaseSettings(
    logging_mode=[20],
    crawling_speed_factor=[1.0],
    default_crawl_delay=[5],
    parallel_process=[i*8 for i in range(1,5)],
    parallel_fetcher=[i*8 for i in range(1,8)],
    iterations=[1],
    fqdn_amount=[16],
    url_amount=[0],
    long_term_part_mode=[enum.LONGPART.none],
    long_term_prio_mode=[enum.LONGPRIO.old_sites_first],
    short_term_prio_mode=[enum.SHORTPRIO.old_pages_first],
    min_links_per_page=[3],
    max_links_per_page=[3],
    lpp_distribution_type=[enum.PAGELINKDISTR.discrete],
    internal_vs_external_threshold=[1.0],
    new_vs_existing_threshold=[1.0],
)
```

Partitionierungsstrategien

```
example_db_settings = dict(
    fqdn_amount=4000,
    min_url_amount=5,
    max_url_amount=5,
    fixed_crawl_delay=10,
)

project_settings = dict(
    name="partition",
    date=datetime.now().strftime("%Y-%m-%d"),
    repetition=1,
)

case_settings = pyd.CaseSettings(
    logging_mode=[20],
    crawling_speed_factor=[1.0],
    default_crawl_delay=[10],
    parallel_process=[16],
    parallel_fetcher=[16],
    iterations=[5],
    fqdn_amount=[50],
    url_amount=[0],
    long_term_part_mode=[
        enum.LONGPART.none,
        enum.LONGPART.top_level_domain,
        enum.LONGPART.fqdn_hash,
        enum.LONGPART.consistent_hashing,
    ],
    long_term_prio_mode=[enum.LONGPRIO.old_sites_first],
    short_term_prio_mode=[enum.SHORTPRIO.old_pages_first],
    min_links_per_page=[5],
    max_links_per_page=[5],
    lpp_distribution_type=[enum.PAGELINKDISTR.discrete],
    internal_vs_external_threshold=[1.0],
    new_vs_existing_threshold=[1.0],
)
```

Priorisierungsstrategien nach Größe der Webseiten

```
example_db_settings = dict(
    fqdn_amount=4000,
    min_url_amount=5,
    max_url_amount=5,
    fixed_crawl_delay=5,
)

project_settings = dict(
    name="prioritization_size",
    date=datetime.now().strftime("%Y-%m-%d"),
    repetition=3,
)

case_settings = pyd.CaseSettings(
    logging_mode=[10],
    crawling_speed_factor=[1.0],
    default_crawl_delay=[5],
    parallel_process=[16],
    parallel_fetcher=[4],
    iterations=[20],
    fqdn_amount=[50],
    url_amount=[0],
    long_term_part_mode=[enum.LONGPART.none],
    long_term_prio_mode=[
        enum.LONGPRIO.large_sites_first,
        enum.LONGPRIO.small_sites_first,
        enum.LONGPRIO.random
    ],
    short_term_prio_mode=[enum.SHORTPRIO.pagerank],
    min_links_per_page=[1],
    max_links_per_page=[1],
    lpp_distribution_type=[enum.PAGELINKDISTR.discrete],
    internal_vs_external_threshold=[0.85],
    new_vs_existing_threshold=[0.95],
)
```

Priorisierungsstrategien nach Aktualität der Webseiten

```
example_db_settings = dict(
    fqdn_amount=4000,
    min_url_amount=5,
    max_url_amount=5,
    fixed_crawl_delay=5,
)

project_settings = dict(
    name="prioritization_age",
    date=datetime.now().strftime("%Y-%m-%d"),
    repetition=1,
)

case_settings = pyd.CaseSettings(
    logging_mode=[20],
    crawling_speed_factor=[5.0],
    default_crawl_delay=[5],
    parallel_process=[10],
    parallel_fetcher=[4],
    iterations=[500],
    fqdn_amount=[20],
    url_amount=[0],
    long_term_part_mode=[enum.LONGPART.none],
    long_term_prio_mode=[
        enum.LONGPRIO.old_sites_first,
        enum.LONGPRIO.new_sites_first,
        enum.LONGPRIO.random
    ],
    short_term_prio_mode=[enum.SHORTPRIO.pagerank],
    min_links_per_page=[2],
    max_links_per_page=[2],
    lpp_distribution_type=[enum.PAGELINKDISTR.discrete],
    internal_vs_external_threshold=[0.85],
    new_vs_existing_threshold=[0.95],
)
```

Erklärung

Hiermit bestätige ich, dass ich die wissenschaftliche Arbeit mit dem Titel *Entwicklung einer Steuerungskomponente zur Priorisierung von Aufträgen für verteilte Webcrawls* selbstständig und ohne fremde Hilfe angefertigt habe. Alle Stellen, welche wortwörtlich oder sinngemäß zitiert oder übernommen wurden, sind auch als solche kenntlich gemacht.

Des Weiteren bestätige ich, dass das Thema meiner wissenschaftlichen Arbeit von mir stammt und nicht von einer schon vorhandenen Arbeit übernommen wurde. Diese Arbeit wurde noch keiner Prüfungsbehörde vorgelegt und nicht veröffentlicht.

Name, Vorname:	Gäbeler, Jens
Matrikelnummer:	70453756
Ort, Datum:	Stuttgart, 19. Juli 2020

Unterschrift: