Diplomarbeit

Eine neue Suchfunktion für die Universität Freiburg

Implementierung einer quellenübergreifenden Suche für die Seiten der Universität Freiburg unter Verwendung von CompleteSearch

Johannes Schwenk

Eingereichte Diplomarbeit gemäß den Bestimmungen der Prüfungsordnung der Universität Freiburg für den Diplomstudiengang Informatik vom 29.11.2003

Institut für Informatik
Lehrstuhl für Algorithmen und Datenstrukturen
Albert-Ludwigs-Universität Freiburg
Freiburg im Breisgau

Autor Johannes Schwenk

Bearbeitungszeit 01. Dezember 2009 bis 16. September 2010

Gutachter Prof. Dr. Hannah Bast

Betreuer Prof. Dr. Hannah Bast

Erklärung

nach §10(7) der Diplomprüfungsordung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Arbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Freiburg, den 14. September 2010

Johannes Schwenk

Zusammenfassung

Diese Arbeit präsentiert die Implementierung einer neuen Suchfunktion für die Webseiten der Universität Freiburg unter Verwendung der Software CompleteSearch als Backend. Dabei wird ein Plugin-System zum einfachen und schnellen Einbinden neuer Quellen in Python entwickelt, welches strukturierte XML-Daten aggregiert. Es wird ein generisches Plugin für Plone-Inhalte vorgestellt, welches das Hinzufügen weiterer Plone-Portale zum Suchindex zusätzlich vereinfacht. Zusammen mit weiteren Plugins für nicht Plone-basierte Quellen wird quellenübergreifende Suche ermöglicht.

Es wird ein Python-Skript als Erweiterung für den Anfrage-Webserver „Zope“ entwickelt, welches die XML-Antwort des Backends verarbeitet und die Suchergebnisse an die Benutzerschnittstelle liefert. Die in JavaScript realisierte Sofort- und Facettensuche erlaubt eine benutzerfreundliche Suche über die indexierten Quellen. Dabei wurde der Effizienz der einzelnen Komponenten durch Messungen der Performance und durchgeführte Optimierungen Rechnung getragen.

Das längerfristige Ziel ist es, die existierende Suchseite abzulösen und die bisherige Verwendung externer Suchanbieter für eine quellenübergreifende Suche über alle Subdomains der Universität verzichtbar zu machen.

Stichwörter: CompleteSearch, Plone, Zope, Suche, quellenübergreifend, Plugin, XML, Python, JavaScript

Abstract

This thesis presents the implementation of a new seach facility for the web pages of the University of Freiburg, using CompleteSearch as a backend. To ensure quick and easy integration of new sources, a plugin system which aggregates structured XML data is developed in Python. A generic plugin for Plone content is presented, which additionally facilitates adding other Plone portals to the search index. Together with other plugins for none-Plone-based sources, a search spanning multiple sources is realized.

To extend the request-webserver ”Zope”, a Python script is developed, which processes the XML response of the backend and delivers the search results to the user interface. The live- and facetted search written in JavaScript, allows a user-friendly search in the indexed sources. Measuring performance and conducting optimizations ensured the efficiency of the individual components.

The long-term goal is to replace the old search page of the University of Freiburg, making the usage of external search providers obsolete by providing a search over all subdomains and various sources.

Keywords: CompleteSearch, Plone, Zope, search, source-spanning, plugin, XML, Python, JavaScript

Danksagung

Hiermit möchte ich Prof. Dr. Hannah Bast und Oliver Trachte vom Webteam der Universität für die ausgezeichnete Betreuung meiner Diplomarbeit danken. Weiter möchte ich meinen Dank an meine Korrektoren Nils Andre, Markus Schwenk, William Duncan und Olivier Förster aussprechen. Für die moralische und kulinarische Unterstützung während der gesamten Zeit danke ich meiner Freundin Anna Heß. Ohne diese Personen wäre die Diplomarbeit nicht möglich gewesen.

Inhaltsverzeichnis

1. Einleitung

Durch das immer schnellere Anwachsen der Datenmengen im Internet stellt sich die Frage, wie diese dem Menschen nutzbringend erschlossen werden können. Durch den Einsatz von immer einfacher zu bedienenden Webanwendungen ist es heute oftmals einer großen Gruppe von Benutzern möglich, Inhalte schnell und ohne technische Kenntnisse auf einer Webseite oder in einer Datenbank zur Verfügung zu stellen.

Gleichzeitig existieren eine Vielzahl technisch unterschiedlicher Systeme, die Daten in ganz unterschiedlichen Formaten vorhalten. Eine gemeinsame Schnittstelle für den einfachen Zugriff und die Verarbeitung der gespeicherten Daten durch Programme fehlt meist. Da das semantische Netz[1] noch in den Anfängen steckt, ist es von großer Wichtigkeit, den Benutzern durch effiziente Suchmaschinen die Möglichkeit zu geben, die Informationsflut des Internet effektiv nutzen zu können.

Während der mehr als dreijährigen Tätigkeit des Autors als studentische Aushilfskraft im Webteam der Stabsstelle *Marketing und Wissensmanagement*, wurde mehrfach die Idee einer neuen Suche für die Seiten der Universität diskutiert. Eine solche Suche sollte im Idealfall alle an der Universität vorhandenen Webseiten und Datenquellen vereinen, und so die Verfügbarkeit bisher sehr schwer auffindbarer Daten über eine effiziente, einfach zu bedienende Suchfunktion erhöhen. Gleichzeitig sollte diese die Verwendung externer Suchmaschinen wie Google, als bisher einziger Möglichkeit eine quellenübergreifende Suche durchzuführen, überflüssig machen.

1.1. Ziel der Diplomarbeit

Die in dieser Diplomarbeit entwickelte quellenübergreifende Suchfunktion soll möglichst viele der Universität Freiburg zugehörige Webseiten durchsuchbar machen. Alle Inhalte sollen über *lediglich ein Eingabefeld* durchsuchbar sein. Der Umfang der indexierten Quellen soll mit geringem Aufwand *erweiterbar* sein, sodass langfristig kein Bedarf mehr an externen Suchmaschinen besteht. Ein Schwerpunkt liegt somit auch in der Schaffung von Mechanismen zur Beeinflussung der *Gewichtung einzelner Begriffe* für spezifische Dokumente, um somit *mehr Kontrolle* über die Reihenfolge der Treffer in den Ergebnissen, und damit letztendlich über deren Qualität, zu haben.

[1] Oft auch als *Web 3.0* bezeichnet.

Die verwendete Suchmaschine soll eine möglichst effiziente Suche ermöglichen, so dass der Benutzer die Ergebnisse bereits während des Tippens angezeigt bekommt, und so in der Lage ist, seine Suchanfrage gegebenenfalls zu revidieren oder zu vervollständigen. Hierzu sollen laufend diejenigen Vervollständigungen des letzten Wortes im Suchfeld angezeigt werden, welche die meisten stark gewichteten Treffer ergeben würden. Weiterhin sollen laufend verfügbare Inhaltskategorien und deren Trefferanzahl für die aktuelle Suchanfrage angezeigt werden. Analog soll auch eine Auswahlmöglichkeit für die Zielsprache des gesuchten Inhalts angeboten werden.

1.2. Aufbau der Diplomarbeit

Die Diplomarbeit ist in fünf Kapitel gegliedert (siehe Abbildung 1.1). Im folgenden Kapitel sollen die *bisherige Suche*, deren Funktionsumfang und Nachteile den geplanten Funktionen der *neuen Suchfunktion* gegenübergestellt werden, und so die Motivation für die im dritten Kapitel beschriebene *technische Realisierung* liefern. Innerhalb dieses Kapitels sind wiederum in jeweils einem eigenen Abschnitt *fünf Komponenten* beschrieben, welche im Zusammenspiel die neue Suchfunktion ergeben. In einem weiteren Abschnitt wird auf die *Performance und Optimierung* der fünf Komponenten eingegangen, bevor abschließend ein *Zwischenfazit* gezogen wird.

Im vierten Kapitel wird das Hintergrundwissen für die *Administration* der Komponenten gegeben. Dabei wird auf die *Netzstruktur* an der Universität eingegangen und die Maßnahmen zur Sicherstellung der *Verfügbarkeit und Aktualität* der Komponenten, sowie auf die *Fehlersuche und -behebung* eingegangen. Ferner finden sich Hinweise zur Verfügbarkeit des *Quelltextes* und dessen *Dokumentation*.

Im fünften Kapitel wird ein *Fazit* gezogen und ein Ausblick auf weitere wünschenswerte Funktionen gegeben.

Im Anhang finden sich vollständige *Klassendiagramme* der Datenerfassung und des Anfrage-Webservers, deren Größe und Ausführlichkeit eine ausgelagerte Platzierung erforderlich erscheinen ließen.

1.3. Konventionen

Einige Sätze, Satzteile und Wörter sind vom normalen Schriftbild abgehoben. Dabei sind Eigennamen, Quelltexte und Quelltext-Fragmente, hervorzuhebende Wörter und Zitate nach den untenstehenden Konventionen formatiert.

- *Kursiv*: Hervorhebungen, wörtliche Zitate, Definitionen und Ersterwähnungen wichtiger Begriffe werden kursiv gesetzt.

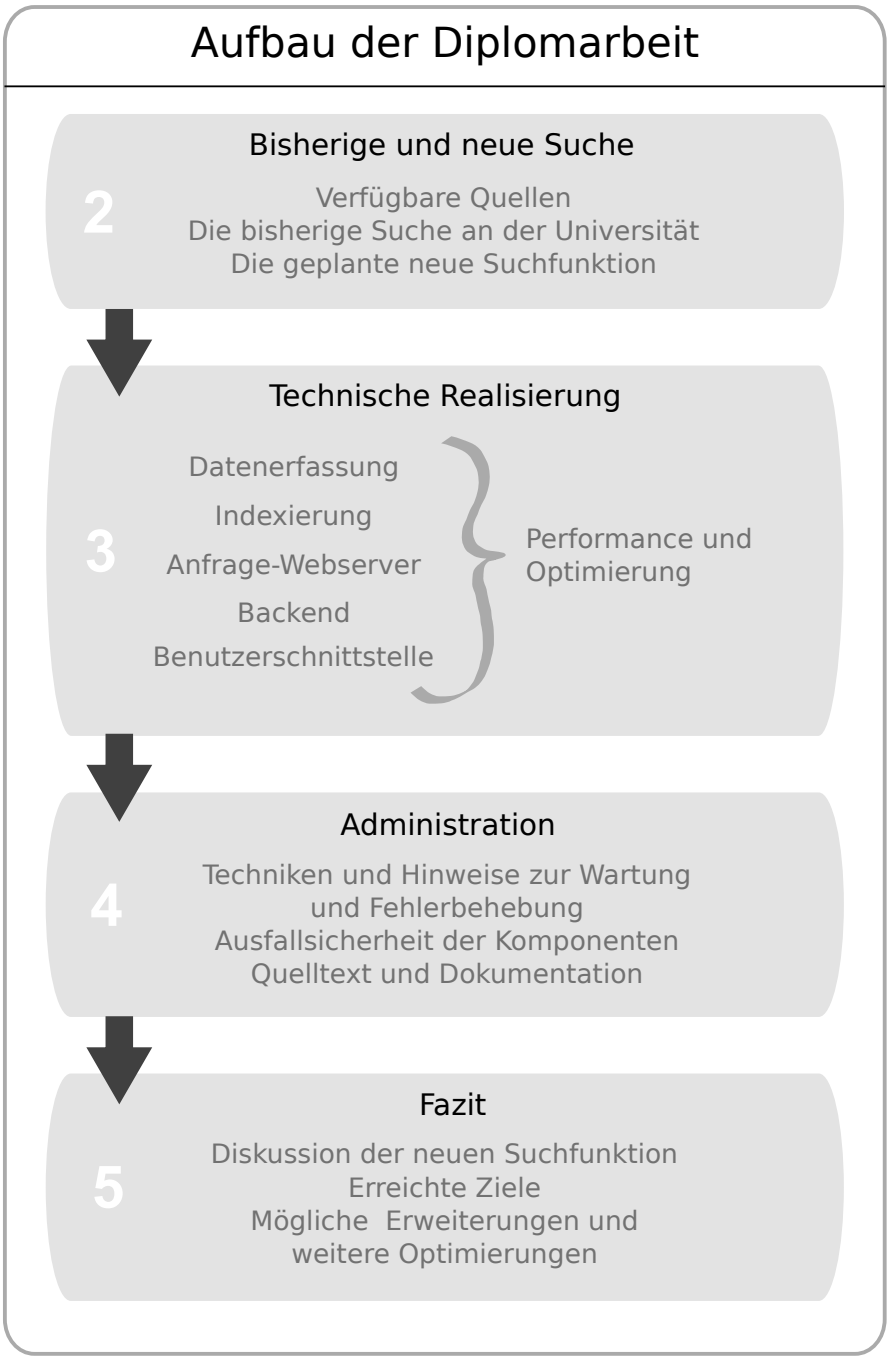

Abb. 1.1.: Aufbau der Diplomarbeit.

- *Schräg*: Eigennamen bei Ersterwähnung, oder an wichtiger Stelle zur Hervorhebung und zentrale Komponenten der neuen Suchfunktion werden abgeschrägt gesetzt.
- **Fettschrift**: Am Anfang von Absätzen werden für diesen zentrale Begriffe fett geschrieben. Ebenso werden Begriffe die eine implizite Aufzählung strukturieren auf diese Weise hervorgehoben.
- `Nichtproportionalschrift`: Dieser Schrifttyp wird für alle Textbausteine verwendet, die Skriptnamen, Parameter oder Quelltext sind.

Da das *LaTeX*-Paket *listings* leider keine UTF-8-kodierte Umlaute verarbeiten kann, sind diese in Quelltexten und Abbildungen von HTML-Fragmenten durch ihre entsprechenden Umschreibungen ersetzt worden.

2. Bisherige und neue Suche

Dieses Kapitel wirft einen Blick auf die vorhandenen Suchmöglichkeiten für die Internetseiten der Universität und motiviert die Implementierung einer neuen quellenübergreifenden Sofortsuche. Dabei werden die Funktionen der vorhandenen Suche beleuchtet und denen der neuen Suchfunktion gegenübergestellt, wobei Vor- und Nachteile herausgearbeitet werden. Weiterhin wird auf die verschiedenen Arten der Datenquellen und die bestehenden Zugriffs- und Suchmöglichkeiten innerhalb der Internetseiten der Universität eingegangen.

2.1. Verfügbare Quellen

Im Wesentlichen lassen sich zwei Arten von Quellen unterscheiden. Zum Ersten sind dort die auf dem frei verfügbaren *CMS*[1] *Plone* basierenden Internetseiten. Zum Zweiten gibt es eine Vielzahl in ihrer technischen Beschaffenheit unterschiedliche Quellen – von *MySQL*-Datenbanken über *Tamino* XML-Servern hin zu *PHP* und *Typo3*-basierten Webseiten. Diese sind im Folgenden unter „Andere Quellen“ zusammengefasst und werden im gleichnamigen Abschnitt unten beschrieben.

2.1.1. Plone-Portale

Seit der Entscheidung zugunsten des CMS Plone für den Internetauftritt der Universität sind insgesamt bereits über 300 Plone-Portale angelegt worden. Viele statische oder PHP-basierte Webseiten wurden dabei migriert. Darunter sind die *zentralen* Seiten der Universität, wie `http://www.uni-freiburg.de`, die Seiten der *Presseabteilung*, *Marketing und Wissensmanagement*, *Alumni*, *Podcasts*, das *Studierendenportal*, die Seiten des *Rechenzentrums*, und viele weitere.

Plone-Portale zeichnen sich als Content Management Systeme gegenüber z.B. individuell erstellten PHP-basierten Seiten ohne Content-Management-Funktion dadurch aus, dass sie erstens zentral durch einen Administrator des Rechenzentrums bereitgestellt und verwaltet werden, und zweitens dadurch, dass sie gut dokumentierte und einfach zu verwendende Möglichkeiten bieten, auf den Inhalt zuzugreifen. Durch die entstehende Homogenität der Webseiten und die zentrale Verwaltung

[1] Englisch: ***C****ontent* ***M****anagement* ***S****ystem*.

können leicht die technischen Voraussetzungen geschaffen werden, um Daten in einem einheitlichen Format zu sammeln. Dies ist wiederum eine gute Voraussetzung, um eine quellenübergreifende Suchfunktion zu realisieren.

Für die vorliegende Arbeit wurden große und repräsentative Portale ausgewählt[2], die den zentralen Seiten unter `http://www.uni-freiburg.de`, der zentralen Universitätsverwaltung oder dem Studierendenportal thematisch oder organisatorisch nahestehen. Durch die zentrale Verwaltung der Portale durch das Rechenzentrum ist es jedoch mit wenig Aufwand möglich, die hier dargestellten Methoden auf weitere Portale anzuwenden.[3]

2.1.2. Andere Quellen

Trotz der weitgehenden Verwendung von Plone für die Webseiten der Universität gibt es viele nicht Plone-basierte Quellen. Dabei kann es sich um MySQL-Datenbanken wie für die Fächerinformationen des Studierendenportals, Tamino XML-Server wie für Forschungsdatenbank, Stellenbörse und Veranstaltungskalender oder extern verwaltete Seiten, wie den Internetauftritt des Studentenwerks oder der Universitätsbibliothek handeln. Die Daten dieser heterogenen Quellen können nur mit erhöhtem Aufwand zentral für eine Suchfunktion vereint werden. So müssen Seiten wie die des Studentenwerks, welche keinen Mechanismus analog zum direkten Auslesen des Inhalts innerhalb eines Plone-Portals verfügen, oder zu denen kein administrativer Zugang besteht, über einen *Webcrawler*[4] indexiert werden[5].

Für die neue Suchfunktion wurden einige Quellen ausgewählt[6], die in der verwendeten Technik unterschiedlich sind, jedoch gleichzeitig den zentralen Seiten und den ausgewählten Plone-Portalen (s.o.) thematisch und netztopologisch nahestehen. Auch für solche „anderen“ Quellen sollte ein Hinzufügen weiterer Webseiten und Datenquellen möglichst schnell und ohne großen Aufwand zu bewerkstelligen sein.

2.2. Bisherige Suche

Bisher kann jedes Plone-Portal nur einzeln mit einer lokalen Suchfunktion durchsucht werden. Dabei gibt es auch Quellen, die erst für diese Suchfunktion verfügbar gemacht werden mussten, wie etwa die Podcasts der Universität. Deren Informationen werden in einer MySQL-Datenbank vorgehalten und daher nicht automatisch von der Plone-eigenen Suchfunktion indexiert.

[2] Siehe Tabelle 3.1.
[3] Siehe Abschnitt 3.2.3.
[4] Englisch, etwa „*Netzkrabbler*“, oft auch als *Spider* (*Spinne*) bezeichnet.
[5] Siehe in Abschnitt 3.2.2 den Absatz über das Plugin für die Seiten des Studentenwerks.
[6] Siehe auch hier Tabelle 3.1.

Für eine universitätsweite Suche wurde bisher ein zusätzliches Suchformular von Google eingebunden. Google bietet die Möglichkeit, die Suche mit dem Parameter `site:uni-freiburg.de` auf diese Domain und die zugehörigen Subdomains einzuschränken. Abbildung 2.1 zeigt die Suchseite mit beiden Eingabefeldern nach dem ersten Aufruf. Weitere Suchmöglichkeiten sind am rechten Rand der Seite verlinkt. Viele dieser Seiten haben jeweils eigene spezialisierte Suchfunktionen.

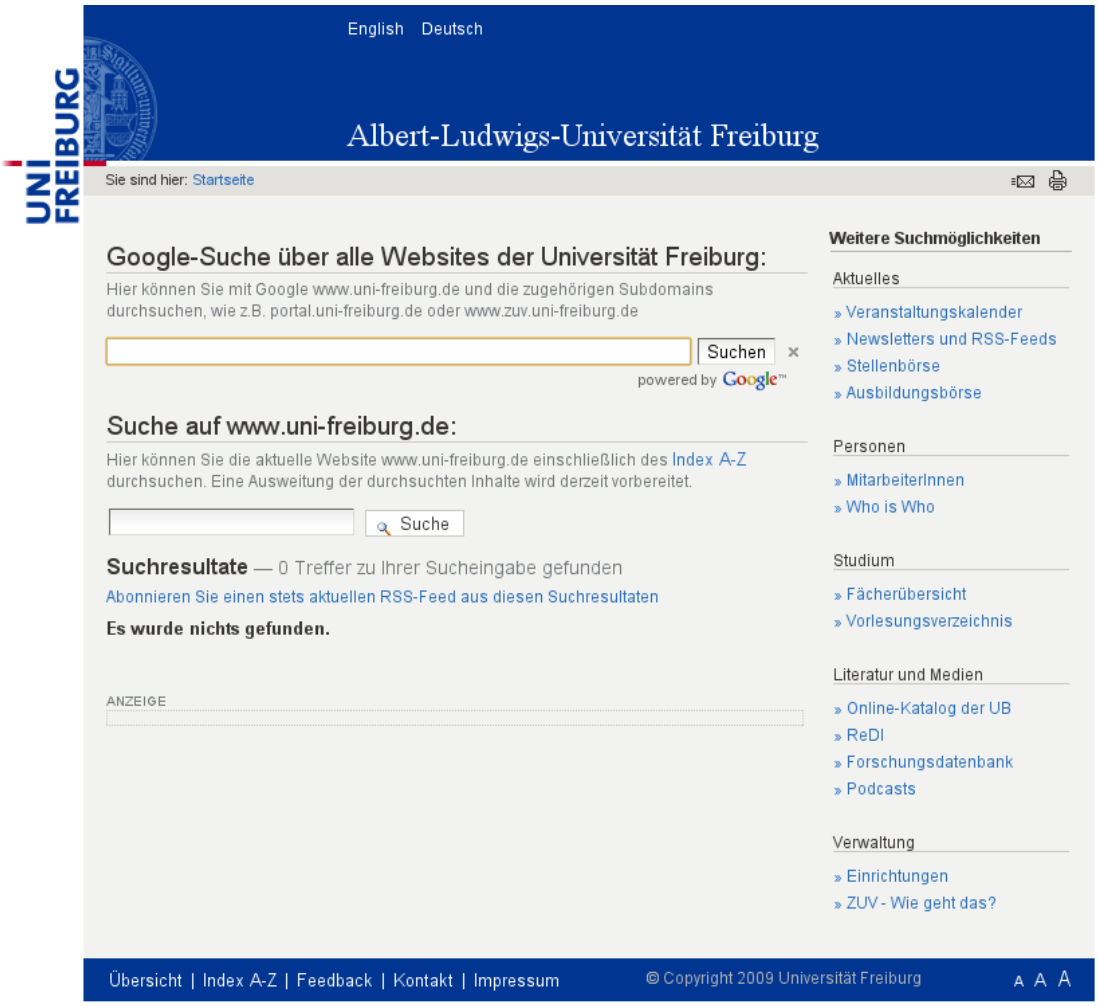

Abb. 2.1.: Die bisher verwendete Suchseite des Plone-Portals unter `http://www.uni-freiburg.de/search`.

Plone führt automatisch eine Volltextindexierung jeder neuen oder geänderten Seite des Portals durch und erlaubt den Zugriff auf die indexierten Daten und Metadaten durch das Plone-Tool `portal_catalog`, woraus auch die mitgelieferte Suche gespeist wird.

2.2.1. Funktionsumfang

Die Suchseite wie in Abbildung 2.1 gezeigt, durchsucht alle indexierten Objekte des Portals im Volltext. Dazu gehören unter anderem auch PDFs, Microsoft Office-, Open-Office-, und PostScript-Dateien.

Eine Suchseite mit erweiterten Optionen kann von der Standard-Suchseite ausgehend erreicht werden. Der Link zu dieser erweiterten Suche, wurde allerdings für das Portal unter `http://www.uni-freiburg.de` deaktiviert. Dies geschah aus der Überlegung heraus, dass den Benutzer die vielen angebotenen Optionen eher verwirren, als ihm zu helfen[7]. Dennoch seien hier die Funktionen der erweiterten Suche der Vollständigkeit halber kurz genannt.

Neben der Volltextsuche kann auch ausschließlich im Titel oder der Beschreibung der Inhalte gesucht werden. Ferner kann der Typ[8] der zu durchsuchenden Elemente, der Autor, das Erstellungsdatum, der Status im Arbeitsablauf[9] und die vergebenen Kategorien[10] eingeschränkt werden. Die Suchfunktion unterstützt die logischen Operatoren *AND* und *OR* zum Verknüpfen zweier Suchbegriffe sowie eine Suche nach Phrasen. Die abgesetzte Suche kann als RSS-Feed abonniert werden, sodass sich einfach eine Benachrichtigungsfunktion für neue Artikel einer bestimmten Kategorie oder eines bestimmten Typs realisieren lässt. Die Ergebnisse der Suche werden nach Relevanz in Prozent absteigend sortiert.

Die Suche ist darüber hinaus als sogenanntes Plone-*Portlet* in Form einer Sofortsuche[11] auf jeder Seite innerhalb des Portals einbindbar. Bei dieser Art der Suche, sind standardmäßig keine Optionen auswählbar, ausgenommen einer Option, um nur den aktuellen Ordner[12] zu durchsuchen.

2.2.2. Beschränkungen

Die bisherige Suchfunktion kann nur lokale Portalinhalte finden, die durch das Plone-Tool `portal_catalog` erfasst wurden. Die Suche mit Google über alle Sub-

[7] Bisher sind keine Beschwerden bezüglich des fehlenden Links beim Webteam der Universität eingegangen. Ein Indiz dafür, dass die Optionen nicht vermisst werden. Die Gründe sind allerdings unklar, und könnten neben einer allgemeinen Ablehnung komplexer Suchformulare auch in der Verfügbarkeit von Google als alternativer Suchmöglichkeit für die Benutzer zu suchen sein. Als ein Beispiel für die erweiterte Suche sei hier auf die Adresse `http://www.rz.uni-freiburg.de/search_form` verwiesen.

[8] Plone-Inhalte haben einen sogenannten `portal_type` – z.B. *Document*, *File*, *Folder*, *Event*.

[9] Auch als Revisionsstatus bezeichnet. Plone regelt die Rechte der Inhalte über solche *Workflows*, die vor allem für die redaktionelle Arbeit hilfreich sind. Artikel durchlaufen so z.B. die Zustände *Entwurf*, *Eingereicht*, *Veröffentlicht*.

[10] Auch *Stichworte* oder *Tags* genannt.

[11] Ergebnisse werden während des Tippens sofort angezeigt.

[12] Ein Ordner ist das Container-Objekt der hierarchischen Inhaltsstruktur eines Plone-Portals und kann Artikel, Dateien und andere Ordner enthalten.

domains unter `uni-freiburg.de` ist jedoch in einigen Bereichen nicht ausreichend, da weder der Prozess der Indizierung gesteuert, noch determiniert werden kann, ab wann die Suchergebnisse aktuell sind. Ferner ist auch eine Gewichtung der Treffer weder direkt beeinflussbar, noch überhaupt brauchbar, da die Funktionsweise von Googles PageRank-Algorithmus nicht bekannt ist. Zum anderen wird so eine nicht notwendigerweise wünschenswerte Abhängigkeit von einem großen Konzern mit oft kritisierten Datenschutz-Praktiken hergestellt, welche das Vertrauen einiger Benutzer negativ beeinflussen könnte.

Die Plone-eigene Suchfunktion liefert keinen Kontext zu den Treffern, sodass sich der Benutzer erst auf den Link zum gefundenen Dokument klicken muss, um zu sehen, an welcher Stelle und in welchem Zusammenhang der gesuchte Begriff enthalten ist. Auch ist es nicht möglich den Kontext des Suchbegriffs schon bei der Suche zu berücksichtigen, also nach bestimmten Wörtern zu suchen, die in einem näheren Umkreis eines anderen auftreten.

Es ist weiterhin nicht möglich fehlertolerant zu suchen. Daher werden keine Dokumente gefunden, die den gesuchten Begriff mit einem Schreibfehler enthalten. Anders herum werden bei falsch geschriebenem Suchbegriff Dokumente die den richtig geschriebenen Begriff enthalten nicht gefunden. Wenn also der Benutzer nach dem Begriff *Lektor* gesucht hat, aber *Rektor* finden will, bekommt er keine (relevanten) Treffer.

Da der Server des Plone-Portals selbst die Suchanfragen bearbeitet, kann es bei intensiver gleichzeitiger Nutzung der Suchfunktion durch viele Benutzer zu einer erhöhten Serverlast, und damit zu einer verminderten Leistung für das Ausliefern der Seiten kommen.

2.3. Neue Suche

Ziel der neuen Suchfunktion ist es, quellenübergreifendes Suchen in den Seiten der Universität zu ermöglichen, und so die Verwendung einer externen Suchmaschine weitestgehend überflüssig zu machen. Wichtig sind dabei auch die Aktualität der Daten und die Möglichkeit einer Einflussnahme auf die Reihenfolge der Ergebnisse.

Weiterhin sollte es mit geringem Aufwand möglich sein, beliebige neue Quellen hinzuzufügen. Dazu wird ein Plugin-System implementiert, welches die Erweiterung um neue Quellen durch Hinzufügen eines neuen, oder die Konfiguration eines bestehenden Plugins erlaubt. Das Aktualisieren der Datenbasis durch das Ausführen der vorhandenen Plugins übernimmt die Komponente der *Datenerfassung*, welche

automatisch zwei mal am Tag gestartet wird, und so die Aktualität[13] der indexierten Einträge gewährleistet.

Die in dieser Arbeit zu implementierende Suchfunktion soll die von Hannah Bast und Ingmar Weber entwickelte Suchmaschine *CompleteSearch* [16] als *Backend* verwenden, welche eine sehr schnelle, fehlertolerante und effiziente Sofort- und Facettensuche[14] in großen Datenmengen ermöglicht [17]. Durch eine Trennung der Suchmaschine in *Backend* und *Anfrage-Webserver* soll eine effiziente Bearbeitung der Suchanfragen und eine Entlastung des Plone-Portals erreicht werden.

2.3.1. Funktionsumfang und Beschränkungen

Die Seite der neuen Suchfunktion soll lediglich ein Suchfeld und jeweils ein Ausklappmenü für die Zielsprache und die zu durchsuchende Kategorie zeigen. Alle Funktionen werden dann über diese Eingabemöglichkeiten realisiert. Dazu gehören Sofortsuche, fehlertolerante Suche, Umgebungs- und Satzsuche, Facettensuche, Synonymsuche und Kategorisierung. Weitere Funktionen sind verfügbar und können bei Bedarf integriert werden.[15]

Die Ergebnisse sollen für jedes Zeichen, dass der Benutzer der Suchanfrage hinzufügt, im Hintergrund abgesandt und das Ergebnis in der Seite eingeblendet werden. Ferner soll ein Neuladen der Seite im Browser weder bei einer Auswahl der Zielsprache oder der zu durchsuchenden Kategorie, noch beim Blättern in den Suchergebnissen notwendig sein.

Die Bewertung der Treffer soll von 100% Relevanz absteigend geschehen und der Wert gut sichtbar für jeden einzelnen Treffer angezeigt werden. Durch die stärkere Gewichtung bestimmter Wörter (wie etwa die vergebenen Stichworte in Plone), soll die Reihenfolge der Treffer beeinflusst werden können.

Grundsätzlich sind alle Funktionen für die der Plone-eigenen Suche auch mit CompleteSearch realisierbar[16]. Boolesche Operatoren zum Verknüpfen von Suchbegriffen sind jedoch nicht vorgesehen, da deren Umsetzung einen erheblichen Implementationsaufwand bedeuten würde und angenommen wird, dass es für den Benutzer ohnehin einfacher ist, die existierende Suchanfrage unter Berücksichtigung der bereits angezeigten Suchergebnisse zu revidieren oder zu verfeinern.

Ein Portlet wie für die bisherige Suche (s.o.) ist für die erste Version der neuen Suchfunktion nicht geplant, wird sich aber rasch realisieren lassen. Auch das Anbieten der Suchergebnisse als RSS-Feed wird vorerst noch nicht umgesetzt. Eine Implemen-

[13] Siehe Abschnitt 4.2.
[14] Siehe Abschnitte 3.4.2, 3.6.2, 3.7.
[15] Siehe Abschnitt 3.4.2.
[16] Siehe auch Abschnitt 3.4.2.

tierung dieser Funktionen ist jedoch für die Weiterentwicklung der Suchfunktion über den Rahmen dieser Diplomarbeit hinaus geplant.

3. Technische Realisierung

Dieses Kapitel beschreibt die technische Realisierung der Suchfunktion. Zunächst wird ein Überblick über die Architektur der Anwendung, die verwendeten Programmiersprachen sowie die Interaktion der einzelnen Komponenten gegeben.

Die Details der Implementierung und Funktionsweise der einzelnen Komponenten werden im folgenden Teil des Kapitels in der Reihenfolge behandelt, in der sie in der praktischen Anwendung zum Einsatz kommen - von der Datenerfassung über die Indexierung, das Backend, das Absetzen der Suchanfragen unter Zope/Plone und das Verarbeiten der Serverantwort, bis hin zur Darstellung für und die Interaktion mit dem Benutzer über die Benutzerschnittstelle.

3.1. Überblick

Dieser Abschnitt beinhaltet einen kurzen Überblick über Zope, Plone und deren Einsatz an der Universität Freiburg. Weiter werden die Komponenten der neuen Suche, deren Zusammenspiel und der Datenfluss von den Datenquellen zum angezeigten Suchergebnis ausführlich beschrieben.

3.1.1. Zope und Plone[1]

Die Universität Freiburg stellt seit Dezember 2004 über das Rechenzentrum der Universität das *Content Management System* (*CMS*) - *Zope/Plone* zum universitätsweiten Einsatz zur Verfügung.

Zope[2] ist ein Webanwendungsserver[3] der ursprünglich von der Firma *Digital Creations*[4] entwickelt wurde. Die ersten Komponenten wurden 1996 im Quelltext veröffentlicht. Darauf basierend wurde die proprietäre Software *Principia* entwickelt. Zope basiert auf dem 1998 offen gelegten Quelltext zu Principia und steht unter der GPL-kompatiblen[4] *Zope Public License* (*ZPL*) zur Verfügung. Die Software ist

[1] Zope wird aktuell in der Version 2.10.8-final und Plone in der Version 3.2.3 eingesetzt. Alle Beschreibungen beziehen sich auf diese Versionen.

[2] Englisch: *Zope:* ***Z*** *****O*****bject** ***P*****ublishing** ***E*****nvironment.*. Das *Z* hat keine spezielle Bedeutung. Zope ist unter `http://zope2.zope.org` zu finden.

[3] Englisch: *Application-Server.*

[4] Heute *Zope Corporation.*

größtenteils in *Python* geschrieben - alle für die Performance kritischen Komponenten jedoch in *C*. Dies gilt auch für die hoch performante und transaktionssichere *Zope Object Database* (*ZODB*), welche als objektorientierte Datenbank der Standardspeicher für Inhalte des Servers ist. Einer der Schlüsselaspekte der ZODB ist die sogenannte *Acquisition*[5], durch welche in Ordnern hierarchisch organisierte Objekte automatisch alle Methoden und Attribute ihrer Eltern erben.[6] Für eine Übersicht der Komponenten eines Zope-Servers siehe Abbildung 3.1.

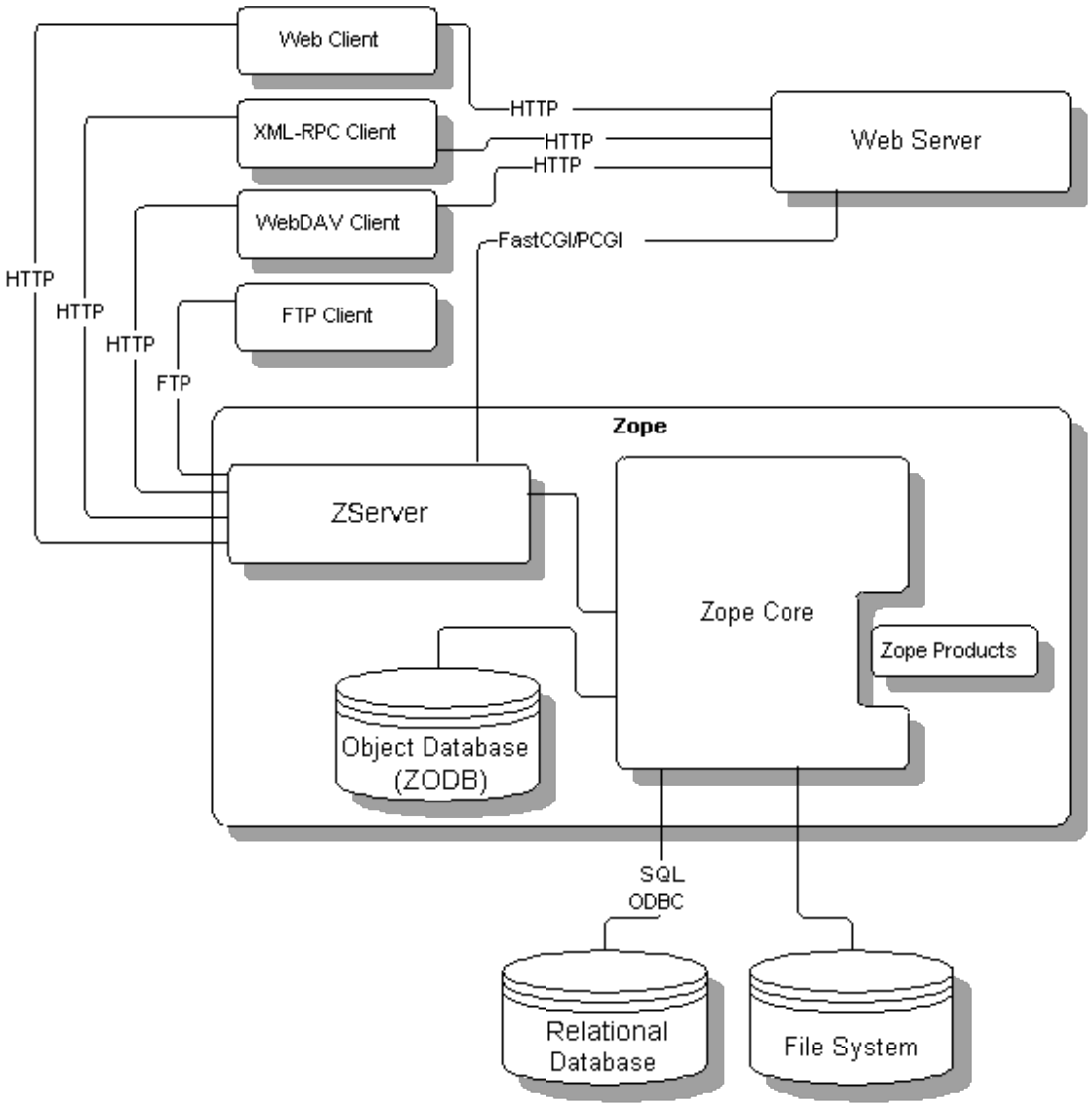

Abb. 3.1.: Komponenten des Zope-Servers. Quelle: [11].

[5] Englisch für etwa: *Übernahme, Aneignung.*
[6] Für technische und erklärende Details siehe [3].

Um die Last des Zope-Servers gering zu halten, kann die ZODB über einen *ZEO*[7]-Server für mehrere Zope-Instanzen (ZEO-Clients) gleichzeitig genutzt werden, siehe [8]. Das Verteilen der HTTP-Anfragen an die ZEO-Clients übernimmt dann ein Load-Balancer. An der Universität Freiburg wird dafür *Pound* eingesetzt; Abbildung 3.2 zeigt den Aufbau wie er an der Universität verwendet wird. Zusätzlich wird der Reverse-Proxy *Squid* eingesetzt um die auszuliefernden Inhalte des Servers zwischenzuspeichern. Dies erfolgt lediglich, um die Hauptlast durch Besucher der Seiten zu minimieren, ist also auf HTTP begrenzt. Der Zugriff über HTTPS erfolgt ohne Zugriff auf Squid. Für eine genauere Beschreibung des Ablaufs eines HTTP-Requests an der Universität siehe auch Abschnitt 4.1 und Abbildungen 4.1,4.2.

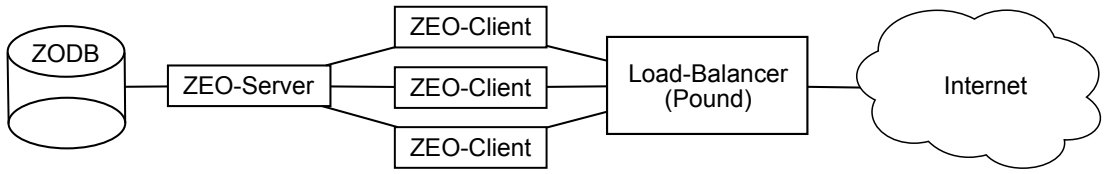

Abb. 3.2.: Server an der Universität mit ZEO-Server, ZEO-Clients und Load-Balancer Pound. Vereinfacht - es fehlt der Reverse-Proxy Squid.

Plone[8] ist ein auf Zope basierendes CMS. Es entstand 2001 als eine Erweiterung von Zopes *Content Management Framework* (*CMF*), um dessen Verwendung zu vereinfachen. Plone wird zusammen mit dem Zope-Server installiert und ermöglicht das Hinzufügen eines *Plone-Portals* in die Objekthierarchie der ZODB. Somit ist ein Plone-Portal ein Objekt innerhalb der Zope-Instanz.

Plone bietet für fast alle Funktionen ein eigenes Backend, sodass nicht auf das Zope-eigene zurückgegriffen werden muss. So können recht einfach Text-, Ton-, Bild- und Video-Inhalte angelegt, bearbeitet und verwaltet werden. Das Bearbeiten textbasierter Objekte kann über integrierte Editoren über einen Internetbrowser auch im sogenannten *WYSIWYG*[9]-Modus oder in externen Editoren per Web-DAV erfolgen. Plone verwendet die Rechteverwaltung von Zope, die eine feinkörnige Vergabe von Rechten für Benutzer und Gruppen auf Objektebene gestattet und bietet anpassbare Workflows, in denen sich sowohl Rechte für Inhaltstypen abhängig von ihrem aktuellen Stadium[10], als auch Aktionen bei Übergang zwischen den Stadien definieren lassen.

[7] ***Z**ope **E**nterprise **O**bjects*
[8] Plone ist unter `http://www.plone.org` zu finden.
[9] Englisch: ***W**hat **Y**ou **S**ee **I**s **W**hat **Y**ou **G**et.*
[10] Typischerweise *Privat*, *Entwurf*, *Eingereicht*, *Veröffentlicht*...

Aktuelle Versionen von Plone haben ein eigenes Framework[11] zum einfachen[12] Erstellen von funktions- und inhaltsbezogenen Erweiterungen. So gibt es eine hohe Anzahl von optionalen *Produkten*[13], die einen großen Umfang unterschiedlicher Anforderungen abdecken.

Für die Darstellung der Inhalte des CMS verwendet Plone sogenannte *Zope Page Templates* (*ZPT*), welche die Template-Sprache *TAL*[14]/*TALES*[15] und die Makro-Sprache *METAL*[16] unterstützen. Mit TAL/TALES kann direkt auf die Objekte der ZODB zugegriffen und deren Attribute und Funktionen verwendet werden. Die Sprache erlaubt unter anderem Schleifen, und es kann sowohl für die Steuerung der Template-Befüllung, als auch für die Erzeugung von Inhalten auf Ergebnisse von Python-Anweisungen zurückgegriffen werden. METAL-Makros erlauben das Wiederverwenden vordefinierter Blöcke, sodass systemweite Änderungen jeweils zentral an einer Stelle durchgeführt werden können. Zudem können Makros sogenannte *Slots* definieren, welche dann im verwendenden Template befüllt werden.[17]

Rechteinhaber an Plone ist die von der Plone-Benutzergemeinde betriebene *Plone Foundation*; Plone selbst ist unter der *GPL Version 2* veröffentlicht.

3.1.2. Architektur

Die Anwendung besteht aus Komponenten, die jeweils einem der folgenden *fünf Bereiche* zugeordnet werden können:

- Datenerfassung
- Indexierung
- Backend
- Anfrage-Webserver
- Benutzerschnittstelle

Abbildung 3.3 stellt den Datenfluss zwischen diesen fünf Komponenten schematisch dar. Datenerfassung und Indexierung laufen hintereinander ab und sorgen für die Verfügbarkeit und Aktualität des Suchindex. Backend, Anfrage-Webserver und Benutzerschnittstelle erlauben im Zusammenspiel das Durchsuchen des Index. Für eine

[11] Das *Archetypes* Framework ist Teil jeder aktuellen Version von Plone.
[12] Mit dem Tool *ArchGenXML* lassen sich aus UML-Klassendiagrammen neue Inhaltstypen generieren.
[13] Die Produkte sind unter `http://www.plone.org/products` zu finden.
[14] ***T**ag **A**ttribute **L**anguage.*
[15] ***TAL E**xpression **S**yntax.*
[16] ***M**acro **E**xpansion **TAL**.*
[17] Für eine ausführlichere Beschreibung von TAL/TALES und METAL siehe [1].

Ersterfassung der Daten und das anschließende Einrichten der neuen Suchfunktion kommen die obigen fünf Komponenten in eben dieser Reihenfolge zum Einsatz.

Während der **Datenerfassung** werden alle Daten gesammelt, die anschließend die Eingabe für die Indexerzeugung darstellen. Hierbei werden verschiedene Quellen nach Daten abgefragt und die Antworten in einem XML-Dokument zusammengefasst. Mit derzeit etwa 320 aktiven Plone-Portalen an der Universität Freiburg ist diese Datenquelle die am häufigsten auftretende. Für die neue Suchfunktion wurden sieben Plone-Portale indexierbar gemacht. Es gibt viele weitere nicht-Plone-Quellen, wobei für die neue Suchfunktion die auf Tamino basierende *Forschungsdatenbank* der Universität und des Universitätsklinikums, die *Stellenbörse* und der *Veranstaltungskalender*, die angebotenen *Studienfächer* an der Universität - welche über eine MySQL-Datenbank in das Studierendenportal eingebunden sind – sowie die auf Typo3 basierte Webseite des *Studentenwerks Freiburg* einbezogen wurden.

Es folgt die **Indexierung**, wobei das zusammenfassende XML-Dokument aus der Phase der Datenerfassung verarbeitet und ein binärer, komprimierter Index erzeugt wird, welcher vom Backend als Datenbasis für die Beantwortung von Suchanfragen verwendet wird.

Das **Backend**[18] verwendet den zuvor erzeugten Suchindex, wartet an einem vorgegebenen Port auf Suchanfragen via HTTP und beantwortet diese gegebenenfalls mit einem XML-Dokument, welches die Treffer zu der Anfrage und gegebenenfalls Vervollständigungen einzelner Suchbegriffe enthält.

Der **Anfrage-Webserver** besteht im Wesentlichen aus einer als *Extension*[19] des Zope-Server realisierte Anwendung, deren Funktionen als sogenannte *Externe Methoden* in einem Plone-Portal verfügbar gemacht werden. Die wichtigste dieser Funktionen ist hierbei jene, um Anfragen an das Backend zu senden. Dabei werden die Parameter für die Suchanfrage von der Benutzerschnittstelle empfangen, durch den *Anfrage-Webserver* aufbereitet und in einer HTTP-Anfrage an das Backend gesendet. Die Antwort des Backends ist ein XML-Dokument, welches der Anfrage-Webserver mittels eines XML-Parsers verarbeitet, in HTML umwandelt und zur Anzeige an die Benutzerschnittstelle zurück sendet.

[18] Im Folgenden des öfteren, nach dem Namen des Server-Programms, auch als *CompletionServer* bezeichnet.

[19] Im Verzeichnis `Extensions` der Zope-Installation installiertes Python-Skript.

Die **Benutzerschnittstelle** ist ein Zope/Plone-Template mit einem Eingabefeld für die Suchanfrage. Sobald sich der Inhalt des Suchfeldes verändert, wird asynchron eine Anfrage an den Anfrage-Webserver geschickt und dessen Antwort anschließend in der HTML-Seite angezeigt. Weiterhin werden dem Benutzer im Moment des Tippens fortlaufend Vervollständigungen mit den höchsten zu erwartenden Trefferzahlen für das jeweils letzte Wort der Suchanfrage vorgeschlagen[20]. Weitere Eingabemöglichkeiten für den Benutzer bestehen im Blättern in den Ergebnissen, dem Auswählen der Zielsprache und der Inhaltskategorie.

3.1.3. Programmiersprachen

Die einzelnen Komponenten der Implementierung sind in verschiedenen Programmiersprachen realisiert. Die Datenerfassung ist in *Python* realisiert worden. Das Backend ist in *C++* geschrieben, der Anfrage-Webserver verwendet Python und die Benutzerschnittstelle verwendet Python, die Template-Sprache *TAL/TALES*, die Makro-Sprache *METAL* sowie *JavaScript* für dynamische Funktionalität.

Python ist die naheliegende Entscheidung, wenn es um die Erweiterung von Zope- oder Plone-Funktionalität geht, da sowohl alle Inhaltstypen wie auch Plone selbst und große Teile von Zope in Python geschrieben sind. Somit ist der Integrations- und Administrationsaufwand für in Python geschriebene Erweiterungen gegenüber Programmen in anderen Sprachen relativ gering, und die Wahl von Python für die Umsetzung der Erweiterungen des Zope-Servers zum Bearbeiten der Suchanfragen naheliegend.

Die Entscheidung für Python für die Datenerfassung beruht auf mehreren Überlegungen: Zum einen ist durch den Einsatz, die Betreuung und Weiterentwicklung der Plone-Portale an der Universität eine *hohe Expertise* in dieser Sprache vorhanden, sodass die Administration und Weiterentwicklung in Python mit geringerem Aufwand erfolgen kann, als dies bei einer Realisierung in anderen Sprachen der Fall wäre. Zum anderen bietet eine Skriptsprache die Möglichkeit der *Fokussierung auf den algorithmischen Teil* der Programmierung, während in hardwarenäheren Sprachen Speichermanagement und Typisierung den Blick auf das Wesentliche erschweren. Daher ist Python-Quelltext leichter zu überschauen - bei weniger Zeilen Quelltext und bei gleicher Funktionalität. Der große *Nachteil* ist jedoch die *geringere Performance* der Skriptsprache Python gegenüber hardwarenäheren Sprachen. Dies liegt an der dynamischen Typisierung und der Notwendigkeit eines Bytecode-Interpreters, die für viele Skriptsprachen kennzeichnend sind. Die geringere Performance bei der Datenerfassung ist im konkreten Fall jedoch unkritisch, da eine Aktualisierung des

[20] Man spricht von *Auto-Suggest*.

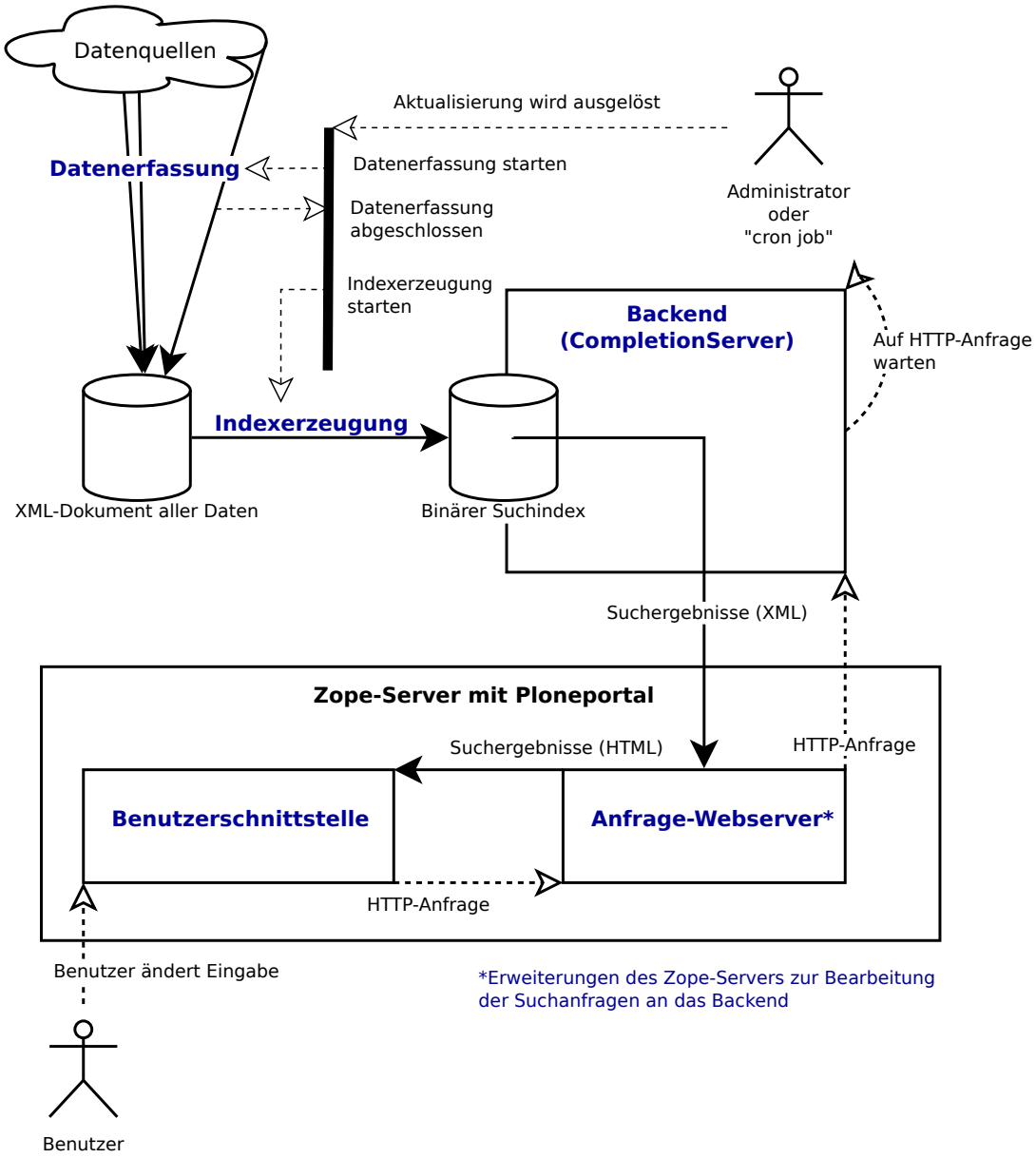

Abb. 3.3.: Datenfluss zwischen den Anwendungskomponenten.

Index zurzeit maximal 45 Minuten dauert und der Index nicht öfter als zweimal am Tag aktualisiert werden soll.

Das *Backend* ist auf Grund der hohen Anforderungen an die Performance in der hardwarenahen Sprache C++ geschrieben.

Die *Benutzerschnittstelle* wird mit Hilfe der Template- und Makro-Sprachen TAL-/TALES und METAL unter Verwendung von Python erzeugt. Die Funktionen zum Absenden der Suchanfrage, das Verfeinern dieser und das Navigieren durch die Ergebnisliste sind in JavaScript unter Verwendung des unter Plone standardmäßig verwendeten *jQuery*[21] realisiert. Zu JavaScript gibt es derzeit *keine praktikable Alternative*, welche vergleichbare Funktionalität und die Kompatibilität mit Internetbrowsern und Betriebssystemen in einem ebenso hohen Maß gewährleisten könnte.

3.2. Datenerfassung

In diesem Abschnitt werden zum einen die Funktionsweise der Komponente zur Datenerfassung erläutert und zum anderen die Möglichkeiten des Hinzufügens zusätzlicher Quellen eruiert.

Hardware: Die Datenerfassung wurde auf dem Rechner `stromboli` des Lehrstuhls durchgeführt. Das verwendete Betriebssystem ist Ubuntu 9.10 Karmic Koala 64Bit. Der Rechner ist mit vier Intel Xeon X5560 2,8 GHz Prozessoren, acht Megabyte Zwischenspeicher und je vier Kernen ausgestattet; der verfügbare Hauptspeicher beträgt 36274 Megabyte. Die Daten liegen auf einem *RAID*[22] welches per *NFS*[23] über Gigabit Ethernet[24] eingehängt ist.

Die Datenerfassung macht zum aktuellen Zeitpunkt Daten aus 93821 Dokumenten über elf Quellen für die Indexierung verfügbar. Die Verteilung der Datensätze auf die Quellen ist Tabelle 3.1 zu entnehmen.

Für die Implementierung dieser Komponente war ein Abwägen zwischen dem Interesse an der Aktualität des Index und der Minimierung der benötigten Ressourcen wichtig. So sollen von der Pressestelle der Universität herausgegebene Pressemitteilungen schnellstmöglich auch über die eigene Suchfunktion gefunden werden können. Dagegen ist es nicht zwingend notwendig, dass Änderungen in der Liste der

[21] *jQuery* ist eine JavaScript-Bibliothek und unter `jquery.com`, sowohl unter der MIT License als auch der GPL verfügbar.
[22] Englisch: ***R**edundant **A**rray of **I**ndependent **D**isks*.
[23] Englisch: ***N**etwork **F**ile **S**ystem*.
[24] Übertragungsrate: 1 Gigabit pro Sekunde.

Quelle	Einträge	Plone	Anmerkung
`forschdb`	84076	Nein	Publikationsangaben der Forschungsdatenbank
`studium`	3552	Ja	Das Studierendenportal
`uni`	1624	Ja	Das zentrale Portal des Webauftritts der Universität (Kopie zu Testzwecken)
`alumni`	1219	Ja	Die Webseite der Alumni
`podcasts`	1078	Ja	Podcast Portal der Uni
`mw`	1063	Ja	Marketing und Wissensmanagement
`studentenwerk`	450	Nein	Studentenwerk Freiburg
`pr`*	400	Ja	Abteilung Presse- und Öffentlichkeitsarbeit
`exzellenz`	180	Ja	Exzellenzinitiative
`vkal`	160	Nein	Veranstaltungskalender
`stb`	19	Nein	Stellenbörse

Tab. 3.1.: Übersicht über die Quellen und deren erfasster Umfang an Datensätzen (Einträgen). Stand: 10. Mai 2010. *Das tatsächliche Volumen des `pr`-Portal liegt weit höher bei etwa 3500 Einträgen, da vergessen wurde, den eigens erstellten Inhaltstyp für die zahlreichen Pressemitteilungen in `remoteSyncQueryXML` hinzuzufügen (siehe auch 3.2.2). Der Fehler ist in der aktuellen Version bereits behoben.

Mitarbeiter/-innen einer Abteilung sofort in den Suchergebnissen nachvollziehbar sind.

3.2.1. Aufbau

Für die Datenerfassung wurden die Python-Pakete `buildxml`, `buildxml.plugins`, und `buildxml.tools` sowie `buildxml.xmlgetter` entwickelt. Abbildung 3.4 zeigt eine Übersicht der vorhandenen Skripte, Pakete, Module und Klassen für die Datenerfassung, ohne Vererbungen und Abhängigkeiten darzustellen. Für ein vollständiges Klassendiagramm siehe Anhang A.1.

Das Skript `getXML.py` enthält das Hauptprogramm, welches die Datenerfassung startet. Es liegt unter `buildxml` und erzeugt bei Aufruf eine Instanz des *Controllers*, welcher wiederum alle *Plugins* sammelt und diese ausführt.

Jede zu erfassende Quelle benötigt ihr eigenes *Plugin*. Dabei wird für Plone-Portale das generische Plugin im Modul `portal` in `buildxml.plugins` mit der Klasse `SyncPlugin_portal` verwendet. Alle anderen Quellen verwenden jeweils ihr eigenes Plugin unter `buildxml.plugins`, wobei der Name des Python-Moduls dem Namen der Quelle entsprechen muss. Das Modul muss eine Klasse mit dem Namen `SyncPlugin_<name>` enthalten, wobei `<name>` der Name der Quelle ist. Alle Daten werden über Instanzen von `XMLEntry` in eine temporäre Datei gespeichert.

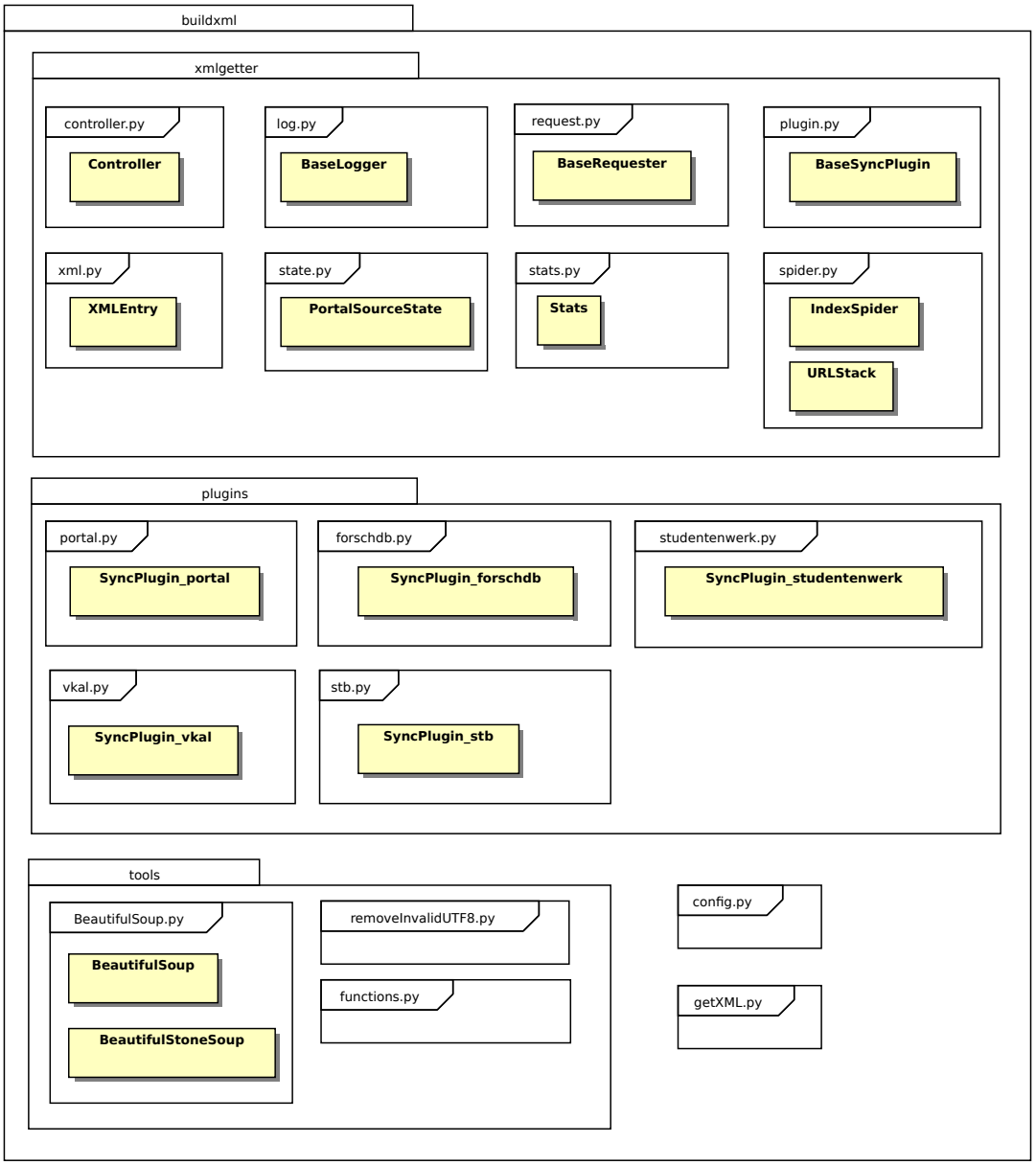

Abb. 3.4.: Übersicht der Pakete und Module für die Datenerfassung.

Dabei müssen eventuell nicht standardkonforme HTML- und XML-Dokumente verarbeitet werden, was derzeit unter der Verwendung der Klassen `BeautifulSoup` und `BeautifulStoneSoup`[25] geschieht. Die angefallenen Daten werden anschließend durch das Skript `removeInvalidUTF8.py` aufbereitet und von `getXML.py` zu einem XML-Dokument zusammengefasst.

Alle Plugins verfügen über eine Instanz von `buildxml.xmlgetter.stats.Stats`, welche statistische Informationen und Meldungen zum Aktualisierungsprozess aufnimmt. Diese Objekte werden nach einer Aktualisierung der Daten durch den Controller eingesammelt und in die Logdatei geschrieben.

Jede Klasse die in die Logdatei unter `buildxml/log/` schreiben soll, ist von der Klasse `buildxml.xmlgetter.log.BaseLogger` abgeleitet und verfügt damit über ein Attribut `logger`, über welches Meldungen in die Logdatei geschrieben werden können.

Das globale Logging-Level, die Registrierung von Plugins und Portalen, sowie viele andere Einstellungen können in `buildxml.config` festgelegt werden (siehe hierzu auch Kapitel 4).

3.2.2. Ablauf einer Aktualisierung

Der Prozess der Aktualisierung wird durch das Skript `getXML.py` gestartet. Das Skript erstellt eine Instanz von `buildxml.xmlgetter.controller.Controller` und ruft dessen Funktion `run()` auf.

Der Controller importiert dann eine Liste aller Portal- und Plugin-Namen aus `buildxml.config`, sucht in `buildxml.plugins` das entsprechende Plugin zu jedem Eintrag und erzeugt eine Instanz von diesem. Dann werden für alle Plugins der Reihe nach deren Funktion `run()` aufgerufen. Die abgerufenen Daten der Plugins sind anschließend in den Dateien mit dem Namensschema `buildxml/tmp/<name>.xml` zu finden und enthalten eine Aneinanderreihung von `XMLEntries`, deren Textform in Abbildung 3.5 zu sehen ist.

Der Aktualisierungsprozess der Plugins kann recht unterschiedlich ausfallen. Der *allgemeine Ablauf* für jedes Plugin ist wie folgt: Die Funktion `run()` des Plugins ruft nacheinander seine Funktionen `_loadState()`, `_getData()`, `_consolidate()`, und `_writeState()` auf. Die Funktion `_consolidate()` kopiert lediglich die temporäre Datei in der `_getData()` die abgerufenen Daten gespeichert hat so, dass sich die Dateiendung zu `.xml` ändert. Alle anderen Funktionen sind in `BaseSyncPlugin` nicht implementiert und können oder sollten von jeder Implementierung eines Plugins überschrieben werden.

[25] BeautifulSoup ist unter der gleichen Lizenz wie Python verfügbar. Siehe [5].

```
<entry>
  <description><![CDATA[Beschreibung]]></description>
  <language><![CDATA[Zweistellige Landeskennung]]></language>
  <created><![CDATA[Datum und Uhrzeit der Erstellung]]></created>
  <url><![CDATA[URL]]></url>
  <portal_type><![CDATA[Typ]]></portal_type>
  <creator><![CDATA[Ersteller]]></creator>
  <modified>
    <![CDATA[Datum und Uhrzeit der letzten Aenderung]]>
  </modified>
  <content><![CDATA[Text des Dokuments]]></content>
  <title><![CDATA[Der Titel]]></title>
  <source><![CDATA[Name der Quelle (des Plugins)]]></source>
</entry>
```

Abb. 3.5.: Textrepräsentation eines `XMLEntry`.

Es folgt die Beschreibung des Aktualisierungsprozesses für die oben genannten Quellen im Einzelnen.

Portal-Plugin: Jedes Plone-Portal, dessen Daten für die Suche erfasst werden sollen, muss ein Python-Skript bereitstellen, welches die Anfragen während der Erfassung beantwortet (im Folgenden mit `remoteSyncQueryXML` bezeichnet[26]). Da das Skript für alle Portale identisch ist, kann die Verfügbarkeit des Skripts durch den Portal-Manager für alle Plone-Portale sichergestellt werden, indem das sogenannte *Basisportal*[27] um `remoteSyncQueryXML` erweitert wird.

Wenn das Portal speziell entwickelte Inhaltstypen enthält, welche ebenfalls indexiert werden sollen, so kann dies erreicht werden, indem der Wert der Variablen `identifier` im Skript um den Inhaltstyp ergänzt wird. Ebenso können bestimmte Inhaltstypen von der Indexierung ausgeschlossen werden, indem sie aus der Liste der `identifier` gelöscht werden.[28]

Um den Hauptspeicherbedarf sowohl des Zope-Servers, als auch den des Rechners für die Datenerfassung gering zu halten, werden die Einträge des Portals inkrementell ausgehend vom ältesten Eintrag in Blöcken zu je drei Einträgen abgerufen. Das Skript `remoteSyncQueryXML` erwartet hierzu drei Parameter: Das *Datum der letzten Anfrage*, die *Anzahl der schon erhaltenen Einträge* und die *Anzahl der zu sendenden Einträge*.

[26] Der Name ist nicht entscheidend. Jeder beliebige Name kann gewählt werden, solange die Änderung im Eintrag des Portals in `buildxml.config` ebenfalls vollzogen wird.

[27] Das Basisportal wird als Template für alle neu einzurichtenden Plone-Portale an der Universität verwendet.

[28] Das Editieren des Skripts kann durch das Zope-Backend direkt im Browser stattfinden – es ist kein direkter Zugriff auf das Dateisystem des Zope-Servers notwendig.

Seit der letzten Aktualisierung nicht geänderte Einträge werden an Kurzeinträgen[29] mit gleicher URL und gelöschte Einträge durch das Fehlen des Eintrags in der Antwort des Skripts angezeigt. Die grundlegende Funktionsweise von `remoteSyncQueryXML` ist dem Pseudocode in Algorithmus 3.1 zu entnehmen.

Das Format der vom Skript gesendeten Einträge muss dem in Abbildung 3.5 gezeigten entsprechen. Zudem muss jeder XML-Tag in einer neuen Zeile beginnen, da der lineare Parser des Portal-Plugins dies als gegeben voraussetzt. Falls keine Einträge mehr auf eine Anfrage zu senden sind, muss die Zeichenkette `END` gesendet werden. Eine solche Terminierung ist notwendig, da das Plugin nicht im Vorhinein wissen kann, wie viele Einträge das Portal beinhaltet, da Änderungen an dessen Inhalt auch während der Anfragen erfolgen können. Das Skript `remoteSyncQueryXML` benutzt den Inhalt des bereits erstellten Indexes des Tools `portal_catalog`[30], sowie die Funktion `SearchableText()`, welche auf den Objekten aufgerufen wird und den Volltext dieser liefert. Dadurch können alle Inhalte extrahiert werden, die über die Plone-eigene Volltext-Suchfunktion gefunden werden. Dazu gehören auch Dokumente im *Portable Document Format* (PDF) von Adobe, sowie Microsoft Office- und OpenOffice-Dokumente.

Das Skript `remoteSyncQueryXML` wird über einen eigens dafür eingerichteten ZEO-Client[31] aufgerufen. Dies ist notwendig, um die ZEO-Clients für den normalen Seitenabruf nicht unnötig zu belasten. So wird das Skript über eine URL der Form

```
http://zope<X>.uni-freiburg.de:<port>/<portal>/remoteSyncQueryXML
```

aufgerufen[32], während die URLs der Einträge in den Suchergebnissen die Form

```
http://www.<portal>.uni-freiburg.de/kurzname-des-eintrags
```

haben sollen. Daher wird in jedem Portal weiterhin ein Eintrag in dem Plone-Tool `portal_properties` unter `remote_query_settings/public_portal_url` benötigt, der die URL des Portals, unter welcher die Seiten normalerweise mit dem Internetbrowser abgerufen werden, beinhaltet.

Für eine Aktualisierung der Daten eines Portals, ruft die Funktion `_getData()` des Portal-Plugins solange die in `buildxml.config` für dieses Portal angegebene URL des Skripts `remoteSyncQueryXML` mit den passenden Parametern auf, bis die terminierende Zeichenkette `END` empfangen wird. In Algorithmus 3.2 ist dieses Vorgehen in Pseudo-Code dargestellt.

In Zeile 8 des Algorithmus wird der Funktionsablauf nach einer fehlerhaften Anfrage für eine Zeit ausgesetzt und dann die Anfrage erneut gesendet, um dem Server Zeit

[29] Format siehe Abbildung 3.6.
[30] Näheres zum Plone-Tool `portal_catalog` im Abschnitt 2.2.
[31] Siehe Kapitel 3.5.
[32] Dabei sind `<X>` die Nummer des Zope-Servers und `<portal>` der Name des Portal, also etwa `studium` für das Studierendenportal.

Algorithmus 3.1 : Der Algorithmus des Skripts `remoteSyncQueryXML`.

Eingabe : *datum* : Das Datum der letzten Aktualisierung
Eingabe : *ab_eintrag* : Sende Einträge ab dieser Nummer
Eingabe : *inkrement* : Die Anzahl der zu sendenden Einträge
Ausgabe : Aneinanderreihung von XML-Einträgen oder 'END'

```
1  pc ← portal_catalog
2  neu_modif ← pc.einträgeGeändertNach(datum, ab_eintrag, inkrement)
3  alt_unmodif ← pc.einträgeGeändertVor(datum, ab_eintrag, inkrement)
4  /* Die Einträge in neu_modif und alt_unmodif sind aufsteigend
      nach Modifikationsdatum sortiert                              */
5  wenn neu_modif = ∅ und alt_unmodif = ∅ dann
6      zurück 'END'
7  sonst
8      /* Im Folgenden ist die Vereinigung ⋃ als der Operator für das
          Anhängen an eine Liste zu lesen.                          */
9      ergebnis ← ∅
10     für jeden eintrag ∈ neu_modif tue
11         xml ← konstruiereXMLEintrag(eintrag)
12         ergebnis ← ergebnis ⋃ xml
13     für jeden eintrag ∈ alt_unmodif tue
14         xml ← konstruiereXMLKurzEintrag(eintrag.url)
15         ergebnis ← ergebnis ⋃ xml
16     zurück ergebnis
```

Algorithmus 3.2 : Die Funktion `_getData()` des Portal-Plugins.

Eingabe : *datum* : Datum der letzten Aktualisierung
Eingabe : *rem_sync_url* : URL von `remoteSyncQueryXML` im Portal
Eingabe : *inkrement* : Anzahl der anzufordernden Einträge
Ausgabe : **Wahr** wenn kein Fehler aufgetreten ist, **Falsch** sonst.

```
1  inhalt ← ''
2  anzahl ← 0
3  solange inhalt ≠ 'END' tue
4      antwort = Falsch
5      solange nicht antwort und nicht maximale Anzahl Versuche erreicht tue
6          antwort ← frageServer(datum, anzahl, inkrement)
7          wenn nicht antwort dann
8              warte kurz
9          wenn antwort dann
10             inhalt ← Inhalt der antwort
11             wenn inhalt ≠ 'END' dann
12                 Schreibe inhalt in temporäre Datei.
13                 anzahl ← anzahl + Anzahl der erhaltenen Einträge
```

zu geben und somit weitere Fehler beispielsweise aufgrund von Überlastung des ZEO-Servers zu vermeiden. Die Länge der Pause und die maximale Anzahl von Wiederholungen sind in `buildxml.config` definiert.

Falls die Aktualisierung *keine* Ersterfassung ist, sendet das Portal nur neue und seit der letzten Aktualisierung geänderte Einträge, alle schon bei einer vorigen Aktualisierung gesendeten Einträge werden durch einen Kurzeintrag, wie in Abbildung 3.6 gezeigt, markiert. Fehlt ein solcher Kurzeintrag, ist der Eintrag als auf dem Portal gelöscht anzusehen. Im Fall einer solchen Folgeaktualisierung existieren bereits Daten von einem vorherigen Lauf in `buildxml/tmp/<name>.xml`. Informationen über das Datum der letzten Aktualisierung, sowie die Positionen der vorhandenen Einträge in der Datei sind in der `_state`-Variablen[33] der Plugin-Instanz zu finden. Ein beispielhafter Ausschnitt der Textrepräsentation eines `PortalSourceState`-Objekts ist in Abbildung 3.7 zu sehen. Die Funktion `_consolidate()` übernimmt die Aufgabe die neu erhaltenen Daten mit den vorhandenen abzugleichen und verfährt hierzu wie in Algorithmus 3.3 skizziert.

Durch das inkrementelle Vorgehen werden Zeit und Ressourcen auf Seiten des Zope-Servers sowie Hauptspeicher auf dem Rechner für die Datenerfassung gespart.

[33] Eine Instanz von `PortalSourceState`.

Ebenso müssen auf diese Weise weniger Daten für eine Aktualisierung über das Netzwerk übertragen werden. Siehe hierzu auch Abschnitt 3.7.

```
<entry>
  <url><![CDATA[<url-des-eintrags-im-plone-portal>]></url>
</entry>
```

Abb. 3.6.: Kurzeintrag aus der Antwort auf eine Anfrage an ein Plone-Portal.

```
{
  http://www.example.com/foo/bar/index.html: [0, 1500],
  http://www.anotherexample.com/bar/foo/file.pdf: [1808, 2323],
}
```

Abb. 3.7.: Beispielauszug eines Teils der Daten der `_state`-Variablen des Portal-Plugins in Python-Notation.

Studienfächer: Eine **Sonderrolle** nehmen die Studienfächer als zusätzliche Quelle innerhalb des Studierendenportals[34] ein. Im Portal ist das Suchen und Anzeigen von Studienfächern und deren Beschreibung über die dynamische Einbindung einer MySQL-Datenbank in die Seite realisiert. Da diese Art Inhalt per se nicht durch das Tool `portal_catalog` erfasst werden kann, wird ein weiteres Python-Skript innerhalb von `remoteSyncQueryXML` aufgerufen, um die zusätzliche Datenquelle zu erfassen. Das Skript hat den Namen `remoteSyncQueryXMLAdditional`[35] und wird, wenn vorhanden, von `remoteSyncQueryXML` automatisch aufgerufen und seine Ausgabe dem Ergebnis angehängt. Durch das Hinzufügen eines Skripts mit diesem Namen ist ein einfaches Erweitern der erfassbaren Inhalte eines Portals möglich. Auch `remoteSyncQueryXMLAdditional` nimmt die selben drei Parameter, wie auch `remoteSyncQueryXML` und gibt ebenfalls `END` zurück, falls keine weiteren Einträge zu senden sind.

Natürlich könnte man die MySQL-Datenbank für die Studienfächer auch separat, also *nicht* über `remoteSyncQueryXMLAdditional` abfragen. Jedoch sind für die

[34] `http://www.studium.uni-freiburg.de`

[35] Auch hier ist der Name nicht ausschlaggebend für die Funktionalität. Für die genaue Funktionsweise ist der Quelltext heranzuziehen, siehe Kapitel 4.

Algorithmus 3.3 : Die Funktion `_consolidate()` des Portal-Plugins.

Eingabe : *empfangene_daten* : Die Datei mit dem Ergebnis der Funktion `_getData()`
Eingabe : *daten* : Datei mit Ausgabe des vorigen Aufrufs der Funktion `_consolidate()`
Eingabe : *daten_info* : Map von URL zu Tupel (Anfang, Ende) in *daten*
Ausgabe : **Wahr** wenn kein Fehler aufgetreten ist, **Falsch** sonst.

```
1  neue_daten ← Neue Datei für konsolidierten Daten
2  neue_daten_info ← ∅
3  für eintrag ∈ empfangene_daten tue
4      wenn eintrag ist ein Kurzeintrag (nicht verändert) dann
5          eintrag_info ← daten_info[eintrag.url]
6          eintrag ← daten[eintrag_info]
7      sonst
8          eintrag ← eintrag ⋃ neuer <source>-Tag mit Quellenname
9      eintrag_start ← Position in neue_daten
10     neue_daten ← neue_daten ⋃ eintrag
11     eintrag_ende ← Position in neue_daten
12     neue_daten_info[eintrag.url] ← (eintrag_start, eintrag_ende)
13     daten ← neue_daten
14     daten_info ← neue_daten_info
```

Integration der Informationen bereits vorgefertigte SQL-Abfragen[36] im Studierendenportal vorhanden, sodass der Aufwand der Implementierung für die Variante über das zusätzliche Python-Skript deutlich geringer ist.

Studentenwerk: Als ein Beispiel eines Plugins für eine Nicht-Plone-Quelle sei hier das komplexere Plugin `studentenwerk` genauer beschrieben. Das Plugin überschreibt lediglich die Funktion `_getData()` der Klasse `BaseSyncPlugin`, da keine Konsolidierung mit früher empfangenen Daten notwendig ist.

Das Plugin verwendet eine Instanz der Klasse `IndexSpider` aus dem Modul `spider` in `buildxml.xmlgetter` um den Dokumentenbaum der Seite des Studentenwerks Freiburg ausgehend von der Sitemap-Seite nach erfassbaren Inhalten zu durchsuchen. Für Testzwecke wurde angenommen, dass alle wichtigen Seiten über die Sitemap mit höchsten zwei Klicks zu finden sein sollten.[37] Daher wird der Baum nur bis zu einer maximalen Tiefe von zwei durchsucht. Derzeit werden nur HTML-Seiten erfasst - Bilder, PDFs, Word-Dokumente und ähnliche werden ignoriert. `IndexSpider` verwendet `URLStack` aus `buildxml.xmlgetter.spider`, sowie mehrere Zwischenspeicher, um Hashwerte des Inhaltes vorzuhalten und somit das mehrfache Erfassen des selben Inhalts zu vermeiden. Weiterleitungen[38] des Servers werden nur bis zu einer Tiefe von eins verfolgt um eventuelle Schleifen zu vermeiden.

Es wird versucht die Serverlast zu minimieren, indem anfangs nur eine *HEAD-Anfrage* per HTTP nach den Kopfdaten des Dokuments gesendet wird. Ist diese erfolgreich, wird zuerst der Inhalt des Header-Feldes `Content-Type` auf Vorkommen in der Liste der erwünschten MIME-Typen – daher in diesem Fall auf Gleichheit mit `text/html` – überprüft. Ist dies gegeben, wird einer eventuellen Weiterleitung einmalig gefolgt und dann (wieder im Falle eines Erfolgs) die Werte der Header-Felder `ETag` und `Content-MD5` überprüft. Ist einer der Werte in einem der *Zwischenspeicher* für *ETags* oder *MD5-Hashes* vorhanden, wird mit der nächsten Seite fortgefahren. Ist die Seite bis hier nicht als Duplikat einer schon abgerufenen Seite identifiziert, wird der Inhalt der Seite über eine GET-Anfrage vom Server abgerufen. Existiert der sha256-Wert[39] des Inhalts nicht im Zwischenspeicher, wird der Inhalt der Seite erfasst, nach Links auf URLs innerhalb des selben Servers durchsucht und diese auf den `URLStack` geschoben.

[36] Sogenannte *Z SQL Methods* [10] sind funktionale Objekte in der ZODB, welche die *Document Markup Template Language* (*DTML*) [2] verwenden können.

[37] Die Sitemap ist unter `http://www.studentenwerk.uni-freiburg.de/index.php?id=272` zu finden, und beinhaltet auf den ersten Blick alles Wichtige.

[38] Englisch: *redirects*.

[39] Englisch: *sha*: ***s**ecure **h**ash **a**lgorithm*. Die Python-Funktion `sha256()` aus dem Modul `hashlib` erzeugt in diesem Fall einen Hashwert der Länge 256 Bits.

Der Aktualisierungsprozess des Plugins startet mit der URL der Sitemap und ist beendet, falls der `URLStack` keine Einträge mehr aufweist.

Die Details der Implementierung aller Plugins, insbesondere für die Quellen **Forschungsdatenbank**, **Stellenbörse** und **Veranstaltungskalender**, welche hier nicht näher beschrieben sind, können dem Quelltext[40] entnommen werden.

3.2.3. Hinzufügen weiterer Quellen

Soll eine weitere Quelle durchsuchbar gemacht werden, gilt es grundsätzlich zwei Fälle zu unterscheiden: Das Hinzufügen eines Plone-Portals und das Hinzufügen andersartiger Quellen.

Hinzufügen eines Plone-Portals: Soll ein Plone-Portal durchsuchbar gemacht werden, muss auf dem Plone-Portal selbst lediglich das Skript `remoteSyncQueryXML` installiert werden und der Eintrag für die Portal-URL in `portal_properties` vorhanden sein[41]. Weiter ist ein Eintrag in `buildxml.config.py` in der Variablen `PORTALS` mit der URL zu `remoteSyncQueryXML` auf dem Portal und dem zu verwendenden Namen für das Portal vorzunehmen. Gegebenenfalls muss noch ein zusätzlicher ZEO-Client zur Entlastung der vorhandenen installiert werden.

Angenommen, die Inhalte des **Studierendenportals** sollen durchsuchbar gemacht werden, dann ist Folgendes zu tun:

1. Installieren des Python-Skripts `remoteSyncQueryXML` auf dem Plone-Portal. Dazu wird das *Zope Management Interface* (*ZMI*) des Portals[42] aufgerufen und ein neues Python-Skript mit dem Namen `remoteSyncQueryXML` und drei Parametern[43] angelegt und das vorgegebene Skript als Inhalt eingefügt.[44]

2. Da das Studierendenportal Inhalte enthält, die nicht durch `portal_catalog` erfasst werden, nämlich die Informationen über die an der Universität Freiburg angebotenen Studienfächer, muss noch ein Skript installiert werden, welches dies übernimmt. Hierzu wird im ZMI analog zum vorigen Schritt das Skript `remoteSyncQueryXMLAdditional` auf dem Portal installiert.

[40] Siehe Abschnitt 4.4.
[41] Siehe Abschnitt 3.2.2.
[42] Unter `http://portal.uni-freiburg.de/studium/manage_main`.
[43] `last_query=None`, `from_idx=0`, `increment=3`. Diese Parameter korrespondieren mit den drei Parametern `datum`, `ab_eintrag` und `inkrement` in Algorithmus 3.1.
[44] Das Skript ist im Quelltext der Datenerfassung zu finden, siehe Abschnitt 4.4. Dieser Schritt kann entfallen, wenn das Skript bereits über das Basisportal vorinstalliert ist.

3. In den `portal_properties` unter `remote_query_settings` wird eine neue Eigenschaft mit dem Namen `public_portal_url` und der öffentlichen URL des Studierendenportals als Wert eingetragen.

4. Gegebenenfalls muss noch ein ZEO-Client erstellt und gestartet werden, sodass das Portal im Folgenden über separate URL und Port[45] abgefragt werden kann, ohne das ein Aufruf der Seiten durch Benutzer beeinträchtigt wird.

5. Eintragen der Quelle in die Python-Variable `PORTALS` vom Typ `list` unter `buildxml.config`.

Mit Ausnahme des Erstellens von `remoteSyncQueryXMLAdditional` im zweiten Schritt, sind die aufgeführten Schritte für alle Portale gleich. Sind keine zusätzlichen Inhalte zu erfassen, entfällt dieser Schritt. Ist zudem `remoteSyncQueryXML` bereits vorinstalliert und ein separater ZEO-Client für die Zope-Instanz bereits vorhanden, - beispielsweise wenn dieser für die Erfassung eines anderen Portals in der selben Zope-Instanz eingerichtet wurde - müssen *lediglich Schritt 3 und 5 ausgeführt* werden.

Hinzufügen anderer Quellen: Soll eine andere Datenquelle eingebunden werden, ist deren Beschaffenheit ausschlaggebend für die konkrete Umsetzung der Einbindung. In jedem Fall jedoch ist ein neues Plugin zu schreiben und einzubinden, was konkret durch folgende Schritte geschieht:

1. Anlegen eines neuen Python-Moduls unter `buildxml.plugins` mit dem Namen des Plugins[46] als Modulname.

2. Erstellen einer Klasse im neuen Modul mit dem Namen `SyncPlugin_<name>` die von der Klasse `BaseSyncPlugin` abgeleitet ist.[47]

3. Überschreiben der Funktion `_getData()` der neu angelegten Klasse. Die Funktion schreibt alle gesammelten Daten in eine Datei unter `TEMP_DIR/<name>` mit der Dateiendung `TEMP_FILE_EXT`, wobei die beiden Variablen `TEMP_DIR` und `TEMP_FILE_EXT` in `config.py` festgelegt sind. Falls eine Webseite durch das Nachvollziehen der Linkstruktur und das Extrahieren der Informationen der verlinkten Seiten erfasst werden muss, kann beispielsweise die Klasse `IndexSpider` aus dem Modul `buildxml.xmlgetter.spider` verwendet werden, analog zum Plugin für das Studentenwerk. Falls keine Konsolidierung notwendig ist, sollten die Daten schon in dieser Phase das Format der textuellen Repräsentation der Klasse `XMLEntry` haben (siehe Abbildung 3.5).

[45] Etwa `http://zope5.ruf.uni-freiburg.de:12345/studium/remoteSyncQueryXML`.
[46] Dieser Name dient gleichzeitig als Quellenname in den Logdateien und E-Mails.
[47] Das Benennungsschema wird verwendet, um die Plugins während der Laufzeit dynamisch an Hand von Konfigurationsparametern nachzuladen.

4. Ist eine Konsolidierung der neuen mit eventuell bereits vorhandenen Daten einer früheren Aktualisierung notwendig, sollte die überschriebene Funktion `_consolidate()` dafür sorgen, dass die durch `_getData()` gewonnenen Daten mit den Daten des letzten Laufs abgeglichen werden. Die konsolidierten Daten sollten anschließend in der Datei `TEMP_DIR/<names>.xml` vorliegen. Ist keine Konsolidierung notwendig (`_consolidate()` ist nicht überschrieben), kopiert die bereits vorhandene Implementierung aus `buildxml.xmlgetter.plugin` die temporäre Datei aus dem vorigen Schritt nach `TEMP_DIR/<names>.xml`.

5. Falls sich das Plugin seinen Zustand (z.B. Datum des letzten Aufrufs) oder Informationen über die vorhandenen Daten für inkrementelles Vorgehen oder Konsolidierung der Daten merken soll, ist dies über das sogenannte *Picklen*[48] eines Zustand-Objektes[49] zu erreichen. Für das Laden und Speichern des Zustands, werden die Funktionen `_loadState()` und `_writeState()` überschrieben, welche von der Funktion `run()` des Plugins vor dem Aufruf von `_getData()`, beziehungsweise nach dem Aufruf von `_consolidate()` aufgerufen werden.

6. Eintragen der Quelle in die Python-Liste `PLUGINS` in `buildxml.config`.

So können potenziell alle Quellen eingebunden werden, die über das Netzwerk erreichbar sind. Python macht den Netzwerkzugriff selbst recht einfach, da es für nahezu jede Art Quelle ein Modul gibt; wie etwa `MySQLdb` für den Zugriff auf MySQL-Datenbanken, oder das Modul `email` für das Verarbeiten von E-Mails aus Verteilerlisten usw.

3.2.4. Aufbereitung der Quelldaten

Nachdem alle erfassten Daten unter `buildxml/temp/<name>.xml` zu finden sind, erfolgt das Zusammensetzen, Bereinigen und Validieren der Daten durch das Skript `getXML.py`. Dieses fügt die temporären Daten aus dem vorigen Schritt zusammen, bettet sie in ein XML-Wurzeldokument samt passender *Document Type Definition (DTD)*[50] ein, und startet `removeInvalidUTF8.py`, um alle ungültigen UTF-8-Multibyte-Sequenzen zu eliminieren sowie unerlaubte Zeichen und Zeichenketten zu entfernen.[51] Schließt ein anschließender Lauf des Programms `xmllint`[52] über die

[48] Englisch: *to pickle*: *einmachen*.
[49] Die Klasse `PortalSourceState` in `buildxml.xmlgetter` kann hierbei als Vorlage verwendet werden.
[50] Siehe Abbildung 3.8.
[51] `removeInvalidUTF8` ist der ursprüngliche Name des Programms, der auch nach Erweiterung der Funktionalität beibehalten wurde.
[52] Die Handbuch-Seite (*manpage*) von `xmllint` gibt Aufschluss über den Funktionsumfang und die Benutzung.

zusammengesetzte Datei `unifr.xml` ohne Fehler ab, wird diese nach `buildxml/out` verschoben und die Phase der *Datenerfassung ist erfolgreich abgeschlossen*.

```
<!DOCTYPE portal [
<!ELEMENT content ( #PCDATA ) >
<!ELEMENT created ( #PCDATA ) >
<!ELEMENT creator ( #PCDATA ) >
<!ELEMENT description ( #PCDATA ) >
<!ELEMENT entry ( description, language, created, url, portal_type,
    creator, content, title, source*, tag* ) >
<!ELEMENT language ( #PCDATA ) >
<!ELEMENT portal ( entry+ ) >
<!ELEMENT portal_type ( #PCDATA ) >
<!ELEMENT title ( #PCDATA ) >
<!ELEMENT url ( #PCDATA ) >
<!ELEMENT source ( #PCDATA ) >
<!ELEMENT tag ( #PCDATA ) >
]>
```

Abb. 3.8.: DTD des Eingabe-XML für den Parser.

3.3. Indexierung

Nach der Datenerfassung folgt die Indexierung. Sie verarbeitet das entstandene valide UTF-8 kodierte XML-Dokument `unifr.xml`[53] weiter. Die dabei entstehende Datenstruktur ist derart optimiert, dass sie Anfragen 3 bis 20 mal schneller beantworten kann als ein herkömmlicher invertierter Index und ist die Grundvoraussetzung für eine effiziente Sofort- und Facettensuche. Abbildung 3.9 zeigt die Verzeichnisstruktur und die für die Indexerzeugung relevanten Dateien.

Dieser Abschnitt gibt einen Überblick über die Implementierung und Funktionsweise des Parsers, sowie die Transformation der erfassten Daten in die komprimierte binären Indexstruktur *HYB*.

3.3.1. Parser

Die in `unifr.xml` vorliegenden XML-Daten mit der in 3.8 gezeigten *DTD* stellen die Eingabe für den Parser dar. Dieser extrahiert aus den XML-Daten eine unsortierte **Wort-** und eine **Dokumentenliste**. Die Wortliste enthält alle Wörter der erfassten Dokumente in Zeilen folgender Form:

[53] Siehe Abschnitt 3.2.4.

```
.
|-- codebase
|   |-- binarysort
|   |   `-- [...]
|   |-- fuzzysearch
|   |   `-- [...]
|   |-- Makefile
|   |-- parser
|   |   `-- [...]
|   |-- server
|   |   `-- [...]
|   |-- synonymsearch
|   |   `-- [...]
|   |-- test-reports
|   |   `-- [...]
|   |-- userinterface
|   |   `-- [...]
|   `-- utility
|   |   `-- [...]
|-- databases
|   `-- unifr
|       |-- codebase -> ../../codebase/
|       |-- Makefile
|       |-- unifr.docs
|       |-- unifr.docs.DB
|       |-- unifr.docs-unsorted
|       |-- unifr.hybrid
|       |-- .unifr.hybrid.build-index-errors
|       |-- .unifr.hybrid.build-index-log
|       |-- unifr.hybrid.from-ascii -> unifr.hybrid
|       |-- unifr.hybrid.prefixes
|       |-- unifr.log
|       |-- unifr.parser
|       |-- unifr.parser.cpp
|       |-- unifr.vocabulary
|       |-- unifr.words-sorted.ascii
|       |-- unifr.words-unsorted.ascii
|       `-- unifr.xml
```

Abb. 3.9.: Verzeichnisstruktur und für die Indexierung relevante Dateien. Abgeschnittene da der Übersicht nicht zuträgliche Teilbäume sind nurch `[...]` gekennzeichnet.

```
[Wort]<tab>[Dokumentnummer]<tab>[Gewichtung]<tab>[Position]
```

Die Dokumentenliste besteht aus Zeilen der Form:

```
[Dokumentnummer]<tab>[URL]<tab>[Text]
```

Dabei ist `<tab>` als Unicode Zeichen `U+0009` (Tabulator) zu verstehen.

Der Quelltext des CompletionServers beinhaltet mehrere Basisklassen zum einfachen Erstellen eines projektspezifischen Parsers. Für dieses Projekt wurde der XML-Parser gewählt und die Klasse `UnifrParser` von `XmlParser` abgeleitet. Diese Subklasse, übernimmt das Parsen der Daten in dem resultierenden C++-Programm. Die Methode `main()` des Programms instanziiert im Wesentlichen ein Objekt der Klasse `UnifrParser` und ruft dessen Methode `parse()` auf dem XML-Dokument `unifr.xml` auf.

Die Klasse überschreibt die Methode `outputDocAndWords()`. Diese Methode wird für jedes XML-Element im Wurzelelement von `unifr.xml` aufgerufen, also in diesem Fall für jeden `<entry>`-Tag[54]. Der Inhalt der Tags ist jeweils über die Funktionen `getItem()` und `getItems()` verfügbar. Aus den Quelldaten in `unifr.xml` werden die Inhalte der Elemente `title` (Titel), `content` (Inhalt), `description` (Beschreibung), `url` (URL), `portal_type` (Typ des Inhalts), `language` (Sprache), `source` (Quelle) und `tag` (Schlüsselwort) verwendet.

Der Eintrag für das Dokument in die **Dokumentenliste** wird mit dem Aufruf der Methode `XmlParser::outputDocument()` mit den Argumenten *Titel*, *URL* und *Inhalt* vorgenommen. Dabei wird die Dokumentnummer automatisch mitgezählt.

Jedes Wort der Felder *Titel*, *Beschreibung*, *URL* und *Inhalt* wird mittels der Methode `XmlParser::outputWords()` in die **Wortliste** eingefügt. Dabei wird die Dokumentnummer, und die Position im Dokument sowie eine Standardgewichtung[55] automatisch vergeben. Alle Satzzeichen und Leerzeichen werden als Worttrenner interpretiert.

Werden vom Autor eines Plone-Inhalts im CMS Kategorien vergeben, so wird angenommen, dass diese Schlüsselwörter für das Dokument sind. Diese Begriffe sind in `<tag>`-Tags zu finden. Für alle diese Tags wird der Inhalt daher mit einer gegenüber „normalen“ Wörtern hundertfach höheren Gewichtung[56] eingefügt.

Die *Sprache* des Eintrags wird als ein Wort der Form `__:language:[xy]:[xy]` eingefügt, wobei `[xy]` dem Inhalt des Feldes, also der zweistelligen Länderkennung entspricht. Für den *Typ* wird ein Wort der Form `__:ptype:[Typ]:[Typ]` einge-

[54] Siehe Abbildung 3.5.
[55] Gewichtung wird hier analog zum englischen *score* verwendet.
[56] Zum aktuellen Zeitpunkt beträgt die Gewichtung für Schlüsselwörter 200.

fügt. Ebenso wird für alle `<source>`-Tags[57] ein Wort `__:src:[Quelle]:[Quelle]` eingefügt.

Da die Zeichen `_` und `:` normalerweise nicht in der Wortliste vorkommen - sie werden als Worttrenner interpretiert - können zum Beispiel mit einer Suche nach `__:language:en:en` alle auf Englisch verfassten Dokumente (und nur diese) gefunden werden. Will man in englischen Dokumenten aus dem Studierendenportal suchen, sucht man nach `__:language:en:en __:src:studium:studium [Suchbegriff]`. Für eine genauere Betrachtung der Suchmöglichkeiten siehe Abschnitt 3.6.2.

3.3.2. Sortierung

Im nächsten Schritt werden die Wortliste und die Dokumentenliste sortiert. Dabei werden die Zeilen der Wortliste nach den Feldern `[Wort]`, `[Dokumentnummer]` und `[Position]` aufsteigend sortiert - im Feld `[Wort]` alphabetisch, numerisch sonst. Die Dokumentenliste wird numerisch aufsteigend nach Dokumentnummer sortiert. Vor der Sortierung wird die *Locale* des Systems per

```
export LC_ALL=POSIX
```

auf die Sortierreihenfolge des *POSIX*[58]-Standards umgestellt. Die Umgebungsvariable `LC_ALL` überschreibt unter anderem die Variable `LC_COLLATE`, welche die Sortierreihenfolge von Textzeichen systemweit festlegt.

3.3.3. Erzeugen des binären Index

Im nächsten Schritt wird der Index in ein binäres Format überführt und komprimiert. Die dabei verwendete Datenstruktur wird mit *HYB* bezeichnet und ist eine Weiterentwicklung der Datenstruktur *invertierter Index* (*INV*).[59]

Ziel ist es, für jedes Zeichen, welches der Benutzer in das Suchfeld eingibt, ohne Verzögerung die am höchsten gewichteten Ergebnisse anzeigen zu können. Gleichzeitig soll dem Benutzer während der Eingabe eine Liste der am höchsten bewerteten Vervollständigungen für das letzte Wort der Anfrage zur Auswahl angeboten werden. Wurde beispielsweise bisher die Anfrage `rektor uni` in das Suchfeld eingetippt, so wären hoch gewichtete Vervollständigungen `rektor universität`, `rektor universitäts`, `rektor universitäten` usw. Dies führt zu Definition 1:

Definition 1 (Vervollständigungsanfrage[60])**.** *Eine Vervollständigungsanfrage ist ein Tupel* (D, W)*, wobei* W *der Wortbereich aller möglichen Vervollständigungen des*

[57] Theoretisch können mehrere Quellen für einen einzelnen Eintrag gesetzt werden, was eine verfeinerte Kategorisierung ermöglicht.
[58] Englisch: *POSIX: **P**ortable **O**perating **S**ystem **I**nterface.*
[59] Für Näheres zur Datenstruktur HYB und ihrer Eigenschaften, siehe [14].
[60] Definition aus [14] übernommen; aus dem Englischen übersetzt.

letzten Wortes der Anfrage und D die Menge der Dokumente für den vorangehenden Teil der Anfrage (bisherige Treffer) ist. Um die Anfrage zu beantworten, muss die Untermenge $W' \subseteq W$ der Wörter berechnet werden, die in mindestens einem Dokument aus D vorkommen, sowie die Untermenge $D' \subseteq D$ der Dokumente, die mindestens eines dieser Wörter enthalten.

Die HYB-Datenstruktur ist nach [14] so konzipiert, dass mit ihr *Vervollständigungsanfragen* im Durchschnitt innerhalb einer Zehntelsekunde beantwortet werden, um dem Benutzer der Suchfunktion instantan Suchergebnisse präsentieren zu können. Der Zeitaufwand für das Beantworten einer Anfrage liegt in $O(n)$, wobei n die Anzahl der Dokumente im Index ist. Dagegen kann die benötigte Zeit bei Verwendung von INV in ungünstigen Fällen n^2 annähern. So ist der HYB-Index im Fall der ungünstigsten Anfrage um den Faktor 15 - 20, und für durchschnittliche Anfragen um den Faktor 3 - 10 schneller als ein entsprechender INV-Index. Ein HYB-Index benötigt nicht mehr *Speicherplatz* als ein korrespondierender INV-Index.[14] Der Speicherplatz für den Index dieses Projekts beläuft sich auf 20% - 30% der Quelldatenmenge - dies sind zum Zeitpunkt des Schreibens 29 Megabyte für den HYB-Index gegenüber 89 Megabyte Ausgangsdaten in `unifr.xml`.

Um die HYB-Datenstruktur zu erzeugen, werden für jedes Wort die invertierten Listen für die Zuordnung der Dokumentnummern, der Gewichtung und der Position erstellt. Die Listen werden jeweils für einen *Wortbereich* erstellt. Der Wortbereich $(abr, bar]$ enthält zum Beispiel alle Wörter zwischen *abr* und *bar*. Wörter, die in diesem Bereich liegen, sind *Abräumen*, *Arbeit* und *Azteken*, während *Abakus* in einem Bereich davor, und *Barium* in einem nachfolgenden Bereich liegen. Die Wortbereiche werden so gewählt, dass die Summe der Länge der Listen für jeden Bereich $\approx c \cdot n$ ist, wobei n die Anzahl der Dokumente ist und $c \approx 0.2 < 1$ konstant. Die Listen der Wortbereiche werden jeweils in binärer Form komprimiert in *Blöcken* gespeichert. Durch diese Einteilung in Blöcke können im Durchschnitt etwa 1 Gigabyte Daten pro Sekunde durchsucht werden. Siehe hierzu und für weitere Angaben zu den Blöcken und ihren Eigenschaften [14].

Ein vereinfachtes *Beispiel*[61] ohne Positionen und Gewichtungen für die Erzeugung eines binären Blocks: Seien A, B, C, und D die vier Wörter des zu verarbeitenden Wortbereichs. Die invertierten Listen für die Dokumentzuordnung sind:

A : $3, 5, 6, 8, 9, 11, 12, 15$
B : $5, 11$
C : $3, 7, 11, 13$
D : $3, 8$

[61] Aus [14] übernommen.

Die *Multimenge der Dokumentnummern* und die *Sequenz der Wörter* sollen nun komprimiert gespeichert werden. Dazu werden die folgenden zwei Zeilen binär kodiert.

3	3	3	5	5	6	7	8	8	9	11	11	11	12	13	15
A	*C*	*D*	*A*	*B*	*A*	*C*	*A*	*D*	*A*	*A*	*B*	*C*	*A*	*C*	*A*

Die Multimenge der Dokumentnummern werden über eine optimale *Lückenkodierung*[62] mit $0 = 0, 1 = 10, 2 = 110, 3 = 111$ kodiert, und die Abfolge der Wörter über eine optimale *Entropiekodierung* mit $A = 0, B = 110, C = 10, D = 111$. Das Resultat sind die folgenden zwei Bitvektoren:

111|0|0|110|0|10|10|10|0|10|110|0|0|10|10|110
0|10|111|0|110|0|10|0|111|0|0|110|10|0|10|0

Dabei sind die |-Zeichen lediglich eingefügt worden um die Lesbarkeit zu verbessern.

Für die neue Suche speichert die HYB-Datenstruktur auch die Liste der *Positionen* der Wörter über eine Lückenkodierung, während die Sequenz mit den zugehörigen *Gewichtungen* unkomprimiert mit gespeichert wird. Die Liste mit den Dokumenteninhalten[63] wird ebenfalls in komprimierter Form in eine separate Datei geschrieben.

Für den Index der neuen Suche zählen die insgesamt 11.714 Blöcke durchschnittlich 695 Wort-Dokument-Paare, bei einer Größe der Index-Datei `unifr.hybrid` von $\approx$ 28 Megabyte ($\approx$ 30% der Ausgangsgröße von `unifr.xml`). Für eine typische Suchanfrage wird nur jeweils ein Wortbereich benötigt, wodurch der Hauptspeicher-Verbrauch minimiert wird. Dies wird unterstützt, indem für sehr häufige Präfixe eigene Wortbereiche erstellt werden [14].

Alle Schritte, vom Parsen über das Sortieren hin zum Erzeugen des binären Index, werden durch den Befehl `make pall` im Verzeichnis des Parsers[64], gestartet. Das Ergebnis sind die Dateien `unifr.docs.DB` mit dem komprimierten Inhalt der Dokumente und `unifr.hybrid` mit der HYB-Datenstruktur. Ebenfalls während der Indexierung werden die Datei `unifr.vocabulary`, welche die alphabetisch sortierten Wörter des Index ohne Duplikate enthält, und die Datei `unifr.hybrid.prefixes` mit einer Liste aller auftretenden Präfixe gespeichert.

[62] Die Lückenkodierungen werden nach dem *Simple-9*-Verfahren durchgeführt, welches in [12] genauer beschrieben ist.
[63] Siehe Abschnitt 3.3.1.
[64] In diesem Fall also `databases/unifr`. Siehe Abbildung 3.9.

3.4. Backend

Nachdem der Index erstellt ist, kann der CompletionServer mit diesem gestartet werden. Der Server liegt typischerweise auf einem anderen Rechner als die Datenerfassung und hat eine möglichst schnelle Anbindung an den Anfrage-Webserver – läuft im Idealfall also auf der selben Maschine wie dieser.[65]

Im Folgenden findet sich eine kurze Beschreibung der Bedienung und der grundlegenden Funktionsweise des CompletionServers, sowie eine Übersicht über bereits eingesetzte und geplante Erweiterungen der Funktionalität.

3.4.1. Funktionsweise und Bedienung

Der CompletionServer kann mit dem folgenden Befehl von der Kommandozeile aus gestartet werden:

```
./startCompletionServer -p 12345 -l unifr.log unifr.hybrid
```

Dabei wird über den Parameter `-p` der Port an dem der Server auf HTTP-Anfragen wartet, mit `-l` der Name der Logdatei und mit dem letzten Argument der Namen der Index-Datei `unifr.hybrid` angegeben. Der Prozess läuft als Dienst im Hintergrund. Ein bereits laufender CompletionServer muss vor einem Neustart nicht beendet werden - der Server erkennt den bereits laufenden Prozess und beendet diesen kurz vor dem Binden an den gegebenen Port. Dadurch wird die Zeit minimiert, in der Anfragen wegen eines Neustarts des Servers abgewiesen werden. Die Ausgabe des Servers in die Logdatei kann mit `tail -f unifr.log` verfolgt werden.

Das Programm `startCompletionServer` startet standardmäßig zwei Instanzen des CompletionServers in jeweils einem eigenen Unterprozess und wartet auf deren Terminierung. Jeder Unterprozess wartet am angegebenen Port auf HTTP-Anfragen und startet für jede dieser Anfragen einen eigenen Thread für deren Beantwortung. Ein Beispiel für eine einfache HTTP-Anfrage an den Server:

```
http://server.test:12345/?q=rektor+kontakt+tel*
```

Dabei ist `+` als URL-kodiertes Leerzeichen zu verstehen. Der für die Anfrage erzeugte Thread des Servers arbeitet die Anfrage in der folgenden Weise[66] ab:

1. Lesen der Anfrage
2. Extrahieren der Parameter
3. Bearbeiten der Anfrage

[65] Im konkreten Fall die unter `zope5.ruf.uni-freiburg.de` erreichbare Maschine mit acht aktiven Zope-Instanzen, darunter auch die Instanz `portal20` mit dem Plone-Portal `uni`, welches die neue Suchfunktion erhalten soll.

[66] Zwecks besseren Verständnisses vereinfacht.

4. Textauszüge aus den Dokumenten für Treffer generieren
5. Senden der Ergebnisse an den Anfrage-Webserver

Der Server sendet ein XML-Dokument als Antwort[67] auf eine Anfrage. Es können GET-Parameter mitgegeben werden um die Art und den Umfang der Serverantwort zu beeinflussen. Hier einige Beispiele:

`h` : Anzahl der Treffer (number of **h**its)
`c` : Anzahl der Vervollständigungen[68] (**c**ompletions)
`f` : Zeige Treffer ab... (show hits starting **f**rom...)
`en` : Anzahl der Textauszüge pro Treffer (von: **n**umber of **e**xcerpts per hit)
`er` : Radius der Textauszüge in Anzahl Wörtern (**e**xcerpt **r**adius)

Die Treffer für eine Anfrage werden innerhalb der Antwort nach *Relevanz* absteigend sortiert[69]. Seien W' und D' wie in Definition 1 angegeben definiert, so wird die Relevanz r_d eines Dokuments $d \in D'$ auf folgende Weise berechnet:

$$r_d(d, W') = \sum_{i=1}^{|W'|} (g(d, w_i)) \tag{3.1}$$

wobei $g(d, w_i)$ die Gewichtungsfunktion ist, welche die Gewichtung der Vervollständigung $w_i \in W'$ für das Dokument d zurückgiebt. Da für die neue Suchfunktion die Gewichtungen der Wörter konstant vorgegeben sind, lässt sich unter der Annahme, dass keine Umgebungssuche[70] durchgeführt wird, die Gewichtungsfunktion vereinfachen zu:

$$r_d(d, W') = \begin{cases} 200 & \text{mind. ein } w_i \text{ kommt in } d \text{ vor und ist ein Schlüsselwort}^{71} \\ 2 & \text{mind. ein } w_i \text{ kommt in } d \text{ vor} \\ 0 & \text{sonst} \end{cases} \tag{3.2}$$

Wird jedoch eine Umgebungssuche[72] mit zwei beteiligten Wörtern a und b durchgeführt, ist also der zu bearbeitende Teil der Anfrage der Form `a..b*`, ist der

[67] Abbildung 3.10.
[68] Für Suchbegriffe die mit * enden.
[69] Dies ergibt eine *Rangfolge*, auch Englisch: *ranking*.
[70] Siehe weiter unten der Absatz über Umgebungs-/Satzsuche, Abschnitt 3.4.2.
[71] Siehe Abschnitt 3.3.1.
[72] Siehe den Absatz *Umgebungs- und Satzsuche* in Abschnitt 3.4.2.

```
<result>
<query>rektor kontakt tel*</query>
<status code="12">OK</status>
<time unit="msecs">0.89</time>
<completions total="29" computed="29" sent="3">
<c sc="2154" dc="158" oc="194" id="402891">tel</c>
<c sc="210" dc="14" oc="35" id="402925">telefon</c>
<c sc="60" dc="5" oc="10" id="402920">telefax</c>
</completions>
<hits total="173" computed="100" sent="1" first="0">
  <hit score="1020" id="97470">
    <title><![CDATA[Der Rektor der Universitaet Freiburg Prof. Dr.
       Hans-Jochen Schiewer]]></title>
    <url>http://www.uni-freiburg.de/universitaet/organisation/
       rektorat/rektor</url>
    <excerpt><![CDATA[<hl idx="0">rektor</hl> Der Rektor der
       Universitaet Freiburg Prof. Dr. Hans-Jochen Schiewer <hl idx
       ="1">Kontakt</hl> Rektor Prof. Dr. Hans-Jochen Schiewer
       rektor@uni-freiburg.de <hl idx="2">Tel</hl>.: +49-761 /
       203-4315 Direktionsassistenz Isolde Schafhausen
       schafhausen@verwaltung.uni ... ]]></excerpt>
    <excerpt>... [there are more matches] ...</excerpt>
  </hit>
</hits>
</result>
```

Abb. 3.10.: Antwort des CompletionServers auf die Anfrage `q=rektor+kontakt+tel*&er=10&en=1&h=1&c=3`.

Rückgabewert der Gewichtungsfunktion g zusätzlich von den Positionen von a in d *abhängig*. Seien B' die Menge der Vervollständigungen für das Wort b und die Positionen eines Worters x innerhalb von d durch die Funktion $pos(d, x)$ gegeben. Dann ist die Relevanz eines Treffers wie folgt definiert:

$$r'_d(d, B', a) = \sum_{i=1}^{|B'|}(g(d, b_i, pos(d, a))) \tag{3.3}$$

Auch für eine Vervollständigung $w \in W'$ wird die Relevanz r_w analog zu der Relevanz der Treffer bestimmt:

$$r_w(w, D') = \sum_{i=1}^{|D'|}(r_d(d_i, \{w\})) \tag{3.4}$$

beziehungsweise

$$r'_w(w, D', a) = \sum_{i=1}^{|D'|}(r'_d(d_i, \{w\}, a)) \tag{3.5}$$

für eine Umgebungssuche, wobei jeweils $d_i \in D'$ und $\{w\}$ die Menge mit w als einzigem Element sind.

Falls bereits für einen eventuell vorhandenen vorangehenden Teil der Anfrage Treffer oder Vervollständigungen vorliegen, werden beim Schnitt der Mengen D und D' sowie W und W' aus Definition 1 die jeweiligen Gewichtungen addiert.

Mit dem Parameter `-s` des CompletionServers kann die Aggregation der Gewichtungen beeinflusst werden. Die oben gezeigten Gleichungen gelten für `-s SSSS`[73], was die Voreinstellung ist. Siehe hierzu die Dokumentation des CompletionServers und dessen Quelltext.

3.4.2. Erweiterbarkeit

Der CompletionServer hat zahlreiche teils optionale Funktionen, von denen im Folgenden einige wichtige kurz beschrieben werden sollen. Für eine genauere Beschreibung der Fähigkeiten des CompletionServers siehe [13, 14, 16] sowie [17]. Alle hier beschriebenen Funktionen können auf das Beantworten einer Vervollständigungsanfrage[74] zurückgeführt werden.

Kategorien und Semantik: Durch das Einfügen von speziellen Wörtern für Inhalte bestimmter XML-Tags in den Index (beispielsweise `beschr:lageplan`) in den Index, können Kategorien vergeben werden. Das Dokument, welches in der Beschreibung

[73] Dabei steht `S` für *Summe*.
[74] Siehe Definition 1, Abschnitt 3.3.3.

das Wort *Lageplan* enthält, kann dann durch die Anfrage `q=beschr:lage*` gefunden werden. Sucht man dagegen mit `q=rektorat+beschr:lageplan`, erhält man eine Liste aller Dokumente, die das Wort `rektorat` enthalten und die Beschreibung `lageplan` haben. Durch Hinzufügen weiterer Prädikate und Relationen in Form von speziellen Wörtern ergibt sich eine *semantische Suchfunktion* sozusagen als Nebenprodukt. Diese Erweiterung wird für die neue Suche bereits genutzt, um vordefinierte Kategorien nach Herkunft der Indexdaten zu vergeben[75]. Semantische Suchanfragen sind jedoch bisher nicht möglich. Für genauere Beschreibungen der Kategorisierungsfähigkeiten und der Funktionen für semantische Suchanfragen siehe [14] und [16].

Facettensuche: Die eingefügten speziellen Wörter ergeben eine Facettenklassifikation der Dokumente im Index. Im Zusammenspiel mit pro-aktiv gestarteten Suchanfragen im Hintergrund kann eine Facettensuche realisiert werden, bei der für jedes getippte Zeichen dem Benutzer zum Kontext passende Kategorien als Mittel zur weiteren Verfeinerung seiner Suchanfrage angeboten werden [14, 15]. Dies wird in der neuen Suche eingesetzt, um kontextbezogen die verfügbaren Kategorien und Sprachen in einem Ausklappmenü zur Auswahl zu stellen. Dafür werden im Hintergrund bei jeder Suchanfrage `<Anfrage>` auch zwei weitere Suchanfragen `<Anfrage> __:language:*` und `<Anfrage> __:category:*` abgesetzt, um die benötigten Informationen zu erhalten. Siehe hierzu auch Abschnitt 3.6.2.

Umgebungs-/Satzsuche: Der CompletionServer unterstützt den Operator `..`[76] für das *lose Verketten* von Wörtern. Sucht man daher etwa nach `max pl*` wird man als am höchsten bewertete Vorschläge die Vervollständigung `platz` erhalten, da sowohl `max` als auch `platz` sehr häufig vorkommen. Sucht man dagegen mit `max..pl*` erhält man als am höchsten gewichtete Vervollständigung `planck`, weil die beiden Wörter direkt hintereinander vorkommen und daher höher gewichtet werden, als Vervollständigungen, bei denen zwischen dem ersten und dem zweiten Wort der Anfrage noch weitere Wörter stehen. Durch Hinzufügen der benötigten Umgebungsinformationen zu den Wörtern des Index erhöht sich die Größe des Index um den Faktor vier bis fünf [14]. Diese Erweiterung steht bereits für die neue Suchfunktion zur Verfügung.

Fehlertolerante Suche: Eine optionale Funktion des CompletionServers ist die *unscharfe*, fehlertolerante Suche[77], mit der sich Wörter trotz einem oder mehrerer

[75] Siehe auch die Beschreibung des Parsers im Abschnitt 3.3.1.
[76] Der Operator kann ohne viel Aufwand durch eine andere Zeichenfolge ersetzt werden.
[77] Englisch: *fuzzy search*.

Tippfehler in der Anfrage, *Tippfehler in den erfassten Dokumenten* oder *ähnlicher Schreibweise* zu einem anderen Wort, finden lassen. So können mit der Anfrage `q=getwitter` auch die Treffer für `gewitter` gefunden werden. Ist anders herum im erfassten Dokument der Titel falsch geschrieben als *Ge**t**witterwarnungen*, so kann es dennoch mit der *korrekt geschriebenen* Anfrage `q=gewitter*` gefunden werden. Ebenso können zusammengesetzte Wörter zu Treffern führen, die nur eines der Wörter enthalten. Anders herum kann die Anfrage `q=gewitter warnungen` Treffer produzieren, die das Wort *Gewitterwarnungen* enthalten (siehe hierzu [17]). Diese Erweiterung ist ebenfalls für die neue Suche in Verwendung.

Synonym-Suche: Eine noch in der Entwicklung befindliche Funktion ist die Suche nach Synonymen der Wörter in der Anfrage. So soll dann beispielsweise die Anfrage `q=semesterferien` auch die Treffer für *vorlesungsfreie Zeit* liefern. Dies wird über eine Synonymdatei mit einer Liste von synonymen Begriffen pro Zeile realisiert. Diese Datei wird vom CompletionServer automatisch in die Beantwortung der Anfragen mit einbezogen. Mit der neuen Suche wird diese Erweiterung ebenfalls bereitgestellt, ist aber noch nicht voll einsatzbereit da noch Anpassungen am der Basisklasse des XML-Parsers nötig sind.

Vervollständigung zu Teilworten und Phrasen: Durch Einfügen von speziellen Wörtern und Phrasen sowie einem geeigneten Operator[78] können auch diese gefunden werden. So könnte zum Beispiel `eigenvektor` eine mögliche Vervollständigung für die Anfrage `normal..vek*` sein. Für Details siehe auch hier [14]. Für die neue Suche ist diese Erweiterung nicht verwendet worden.

Verbundoperationen: Nehmen wir an, der Index enthält Daten über Politiker, welche zusätzlich über ein spezielles Wort `__:politiker:` ausgezeichnet sind. Durch das Schneiden der beiden resultierenden Listen mit *Vervollständigungen* für beispielsweise die Anfrage `deutsch kanzler __:politiker:politiker:` und die Anfrage `audienz papst __:politiker:politiker:` erhält man eine Liste der Politiker, welche eine Audienz mit dem Papst hatten. Dies lässt sich für beliebige Verbundoperationen, wie von relationalen Datenbanken bekannt, und, mit einer kleinen Anpassung, auch für linken/rechten oder natürlichen Verbund realisieren. Siehe hierzu ausführlich in [16]. Diese Funktion wird in der neuen Suche nicht benötigt.

[78] Sinnvoll ist hier die Verwendung des bereits vorhandenen Umgebungsoperators `..` (s.o.).

3.5. Anfrage-Webserver

Die Erweiterungen des Zope-Servers zur Bearbeitung der Suchanfragen an das Backend, im Folgenden auch kurz *Anfrage-Webserver* genannt, leiten von der Benutzerschnittstelle im Hintergrund gesendete Suchanfragen an den CompletionServer weiter und wandeln dessen XML-Antwort in HTML-Quelltext um und dienen somit dem Zope-/Plone-Server als Interface zum Backend.

Im Folgenden wird eine kurze Übersicht über die vorhandenen Klassen und Funktionen des Anfrage-Webservers gegeben, sowie deren Arbeitsweise umrissen.

3.5.1. Aufbau

In den folgenden Beispielen und Erklärungen wird stark vereinfacht und nur die für das Verständnis der Funktionsweise wesentlichen Parameter werden beschrieben. Alle Parameter der Funktionen sowie eine ausführlichere Beschreibung ihrer Funktionsweise sind dem Quelltext und der Dokumentation zu entnehmen.[79]

Der Anfrage-Webserver besteht aus dem Modul `complete`, welches die in Tabelle 3.2 gezeigten *Klassen* bereitstellt. Weiterhin stellt das Modul unter anderem die in Tabelle 3.3 aufgelisteten *Funktionen* zur Verfügung. Das Modul enthält weitere Funktionen, die nur intern[80] verwendet werden und daher hier nicht explizit aufgezählt sind. Die in der Tabelle aufgelisteten Funktionen werden als *External Method*s mit dem gleichen Namen unter `/portal_skins/custom/csearch` eingebunden. Solche *Externe Methoden* stellen Python-Skripte im Dateisystem des Zope-Servers zur Verfügung und sind notwendig, da innerhalb der ZODB abgelegte Skripte nur eine eingeschränkte Python-Umgebung zur Verfügung haben und daher benötigte Module wie beispielsweise `urllib2` oder `re` und selbst einige Zope-eigene Module nicht importiert werden können. Diese Module werden jedoch benötigt um HTTP-Anfragen an das Backend abzusetzen und das Ergebnis in sinnvoller Weise verarbeiten zu können. Weitere Informationen zu Externen Methoden sind in [9] zu finden.

Das Modul wird im Verzeichnis `Extensions` der Zope-Installation abgelegt. Um beispielsweise die Funktion `query()` im Portal verfügbar zu machen, muss anschließend im ZMI nur noch ein Objekt mit dem Typ *External Method* angelegt, sowie eine *ID* (idealerweise auch `query`), der *Module Name* `complete` und der *Function Name* `query` angegeben werden. Nach erfolgreichem Anlegen kann die Funktion über `context.csearch.query`[81] aufgerufen werden.

[79] Siehe hierzu den Abschnitt 4.4 sowie im Anhang das vollständige Klassendiagramm des entwickelten Python-Moduls in Abbildung A.2.
[80] Sie sind also nicht als Externe Methoden zugänglich gemacht.
[81] Wenn die Externe Methode unter `/portal_skins/custom/csearch` liegt.

Name der Klasse	Beschreibung
`CSLogger`	Verwendet ein Singleton-Pattern um *eine* Instanz eines Loggers für alle anderen Klassen und Funktionen bereitzustellen.
`CSParseHandler`	Wird verwendet um die XML-Antwort des Backends mit Hilfe von `xml.parsers.expat` zu verarbeiten.
`CSQuery`	Kapselt die Anfragedaten, welche zum Backend geschickt werden.
`CSHit`	Kapselt die Daten eines Treffers (Dokument) in den Suchergebnissen.
`CSCompletion`	Kapselt die Daten einer Vervollständigung.
`CSRequest`	Ein Objekt dieser Klasse kann verwendet werden um eine Anfrage an das Backend zu stellen. Ihr Konstruktor erwartet eine Instanz eines `CSQuery`.
`CSResponse`	Ein Objekt dieser Klasse wird nach einer erfolgreichen Anfrage an das Backend erzeugt und ist als Attribut der `CSRequest`-Instanz verfügbar. Das Objekt enthält unter anderem alle Treffer und Vervollständigungen.
`CSTimingInfo`	Die Daten für das Timing der Anwendung werden in einem Objekt dieser Klasse zusammengefasst.

Tab. 3.2.: Übersicht über die bereitgestellten Klassen des Moduls `complete`.

Name der Funktion	Beschreibung
`query()`	Gibt ein HTML-Fragment mit den Treffern einer Suchanfrage zurück.
`getNumberOfDocuments()`	Gibt die Anzahl der Dokumente im Index zurück.
`getCurrentLanguage()`	Gibt die zweistellige Länderkennung der aktuell verwendeten Benutzersprache zurück.
`getLanguageMenuHTML()`	Gibt ein HTML-Fragment mit den im Kontext der Suchanfrage verfügbaren Sprachen zurück.
`getCategoriesMenuHTML()`	Gibt ein HTML-Fragment mit den im Kontext der Suchanfrage verfügbaren Kategorien zurück.
`getCompletions()`	Gibt die Liste der für den Kontext der Suchanfrage verfügbaren Vervollständigungen zurück.
`logFinalMessage()`	Schreibt in die Logdatei `complete.log` unter `var/log/`.

Tab. 3.3.: Übersicht über die als External Methods bereitgestellten Funktionen des Moduls `complete`.

Einige als *Externe Methode* eingebundene Funktionen werden im Plone-Portal durch ein Python-Skript gekapselt, welches gegebenenfalls den Kontext des aufrufenden Templates und die anonymisierte IP-Adresse[82] des Benutzers übergibt. Der Kontext, das Python-Objekt `context` wird benötigt, um die Übersetzung der für die Anzeige bestimmten Zeichenketten in die vom Benutzer gewählte Sprache vorzunehmen, da zum Übersetzen das Python-Skript `t`[83] verwendet wird und dieses über `context.csearch.t` angesprochen werden muss. Dieses Skript bedient sich wiederum der Funktion `getCurrentLanguage()`, um die zweistellige Länderkennung der aktuell gewählten Benutzersprache zu erhalten. Die anonymisierte IP-Adresse wird für statistische Auswertungen und die Fehleranalyse gespeichert.

3.5.2. Anfragen an das Backend

Der Ablauf einer Anfrage folgt grundsätzlich dem in Abbildung 3.11 gezeigten Schema. Die Anfrage wird dabei gesendet, wenn auf das Attribut `response` des `CSRequest`-Objekts zugegriffen wird[84]. Das Attribut stellt die empfangene XML-Antwort[85] nach dem Verarbeiten mit dem XML-Parser aus dem Modul `xml.parsers.expat`

[82] Letztes Oktett der Adresse ist durch `XXX` ersetzt. Beispiel: `132.230.62.XXX`.
[83] Siehe den Abschnitt 3.6.1.
[84] Siehe Zeile 3 in Abbildung 3.11.
[85] Siehe Abbildung 3.10.

unter Verwendung von `CSParseHandler` als ein Objekt der Klasse `CSResponse` zur Verfügung.

```
1 query = CSQuery(q, show_from=0)
2 request = CSRequest(query)
3 response = request.response  # Anfrage wird gesendet, die Antwort
4                              # verarbeitet und als Objekt der
5                              # Klasse CSResponse verfuegbar.
```

Abb. 3.11.: Absetzen einer Anfrage an das Backend.

Der Parameter `q` ist die Zeichenkette der Anfrage und korrespondiert mit dem GET-Parameter `q` des CompletionServers[86]. Über `show_from` wird der Server angewiesen die angegebene Anzahl Treffer zu überspringen und erst ab dem `show_from`'ten Treffer zu senden, korrespondiert also mit dem GET-Parameter `f` des Servers.

Für jedes durch den Benutzer eingegebene Zeichen wird – nach 400 Millisekunden Inaktivität des Benutzers – eine Anfrage an den CompletionServer gestellt. Dies geschieht über die Funktion `query()`, deren vereinfachter Ablauf in Abbildung 3.12 zu sehen ist. In Zeile 4 wird durch den Aufruf von `getRowsHTML()` ein HTML-Fragment

```
1 query = CSQuery(q, show_from)
2 request = CSRequest(query)
3 response = request.response
4 rows_html = getRowsHTML(query, response)
5 [...]
6 return html
```

Abb. 3.12.: Stark vereinfachter Ablauf der Funktion `query()`.

mit den Ergebnissen aus `response` erstellt. Die Funktion wird ebenfalls von `complete` bereitgestellt und ist im Abschnitt 3.5.4 näher beschrieben. Das resultierende HTML-Fragment[87] wird von einer JavaScript-Funktion der Benutzerschnittstelle anschließend in die Suchseite eingefügt. Siehe hierzu weiter Abschnitt 3.6.2. Der Rückgabewert ist das Attribut `hits_total` der Serverantwort in `response`, dessen Wert die gewünschte Gesamtanzahl der Dokumente im Index ist.

Die Funktion `getNumberOfDocuments()` wird verwendet, um die Gesamtanzahl der Dokumente im Index des Servers zu erhalten. Sie sendet auf die in Abbildung 3.13 dargestellte Weise eine Anfrage mit leerem Parameter `q` an den CompletionServer.

[86] Siehe Abschnitt 3.4.1.
[87] Siehe Abbildung 3.17.

```
query = CSQuery(u'', show_from=0)
request = CSRequest(query)
response = request.response  # Objekt der Klasse CSResponse
return response.hits_total
```

Abb. 3.13.: Vereinfachter Ablauf der Funktion `getNumberOfDocuments()`.

Die Funktionen `getCategoriesMenuHTML()` und `getLanguageMenuHTML()` senden jeweils eine Anfrage an das Backend, indem sie die Zeichenkette der Suchanfrage um die speziellen Wörter `__:cat:*` bzw. `__:language:*` erweitern. Abbildung 3.14 zeigt ein Beispiel für `getLanguageMenuHTML()`. Beide Funktionen iterieren anschließend

```
query = CSQuery(u'biologie __:cat:studienfach:_studienfach __:
    language:*', show_from=0)
request = CSRequest(query)
response = request.response  # Objekt der Klasse CSResponse
[...]
```

Abb. 3.14.: Suchanfrage der Funktion `getLanguageMenuHTML()` im Kontext der bereits eingegebenen Suchanfrage nach `biologie` in der Kategorie der Studienfächer.

über die Liste der Vervollständigungen `response.completions` und generieren ein HTML-Fragment, welches durch JavaScript-Anweisungen der Benutzerschnittstelle eingefügt und durch CSS-Anweisungen zu einem Ausklappmenü für die Auswahl der zu durchsuchenden Sprache, beziehungsweise Kategorie, wird (siehe Abschnitt 3.6.1).

Die gewünschten Kategorien können über einen Eintrag in `CompleteSearch` in den `/portal_properties` in der Positiv-Liste `categories` sichtbar gemacht werden. Kategorien, die nicht in dieser Liste sind, werden nicht zur Auswahl im Ausklappmenü für die Kategorien angezeigt.

Die Funktion `getCompletions()` arbeitet analog zu den zwei gerade beschriebenen Funktionen und wird von der Benutzerschnittstelle aufgerufen, um die aktuelle Liste der möglichen Vervollständigungen des letzten Wortes im Suchfeld sowie deren Trefferzahlen zu erhalten. Diese Liste wird dann unterhalb des Suchfeldes zur Auswahl durch den Benutzer angeboten. Siehe hierzu Abschnitt 3.6.2 und Abbildung 3.21.

3.5.3. Verarbeiten der Serverantwort

Die Klasse `CSParseHandler` wird verwendet, um die XML-Antwort des Backends[88] zu verarbeiten und ein Objekt der Klasse `CSResponse` zu füllen. Abbildung 3.15 zeigt die vorgesehene Verwendung der Klasse.

```
1 response = urlopen(request)
2 cs_response = CSResponse(response)
3 handler = CSParseHandler(cs_response)
4 parser = xml.parsers.expat.ParserCreate()
5 parser.StartElementHandler = handler.start_xml_element
6 parser.EndElementHandler = handler.end_xml_element
7 parser.CharacterDataHandler = handler.data_xml_element
8 parser.Parse(cs_response.data, 1)
```

Abb. 3.15.: Verwendung von `CSParseHandler`.

Die Funktionen `start_xml_element()`, `end_xml_element()` werden vom XML-Parser aufgerufen, wenn ein XML-Element beginnt oder endet. Für den Inhalt eines Elements wird `data_xml_element()` aufgerufen. Da die XML-Antwort des Servers keine rekursiv geschachtelten Elemente enthält, kann das Verarbeiten ohne neu definierte Stacks geschehen. Abbildung 3.16 zeigt als Beispiel das Vorgehen für die Treffer-Elemente (`<hit>`) der Antwort.

3.5.4. Erzeugen der HTML-Antwort

Alle HTML-Fragmente werden über Zeichenketten-Templates unter Verwendung der Klasse `Template` aus dem Modul `string` erzeugt. Dabei sind die Template-Variablen der Form `${<Bezeichner>}` und können durch den Aufruf der Funktion `substitute()` der Klasse `Template` durch Variablenwerte ersetzt werden. In einer interaktiven Python-Shell sieht das beispielsweise so aus:

```
>>> from string import Template
>>> template = Template(u'<a href="${url}" title="${title}">
...     ${title}</a>')
>>> html = template.substitute(url=u'http://www.uni-freiburg.de',
...     title=u'Universitaet Freiburg')
>>> print html
<a href="http://www.uni-freiburg.de"
...     title="Universitaet Freiburg">Universitaet Freiburg</a>
```

[88] Siehe Abbildung 3.10.

```
def start_xml_element(self, name, attrs):
    if name == 'hit':
        self._inside_hit = True  # Instanz-Variable, welche
                                 # anzeigt, dass sich der
                                 # Parser nun innerhalb
                                 # eines hits befindet.
        self._hit = CSHit()  # Erzeugen eines neuen
                             # CSHit-Objekts fuer den Treffer.
        self._hit.hid = int(attrs['id'])
        self._hit.score = int(attrs['score'])  # Gewichtung.
    elif self._inside_hit:
        # Variablen fuer untergeordnete Elemente belegen.
        [...]
    # Andere Elemente verarbeiten.
    [...]

def data_xml_element(self, data):
    if self._inside_hit:
        # Titel, URL und Textauszuege verarbeiten.
        [...]
    # Andere Elemente verarbeiten.
    [...]

def end_xml_element(self, name):
    if name == 'hit':
        self._inside_hit = False  # Das Treffer-Element wurde
                                  # verlassen.
        self._response.hits.append(self._hit)
        # Der Liste der Treffer des CSResponse-Objekts wurde
        # ein Treffer hinzugefuegt.
    # Ander Elemente verarbeiten.
    [...]
```

Abb. 3.16.: Verarbeiten eines Treffers in der XML-Antwort des Backends mit `CSParseHandler`.

In der Funktion `getRowsHTML()` wird das HTML-Fragment zur Anzeige der Treffer eines Suchergebnisses erzeugt. Dazu wird ein Template (`_row_template`) mit den Daten aus `response.hits` gefüllt. Es wird über alle Treffer iteriert und alle erforderlichen Template-Variablen in einem Wörterbuch[89] (`params`) gesammelt. Die aus dem Befehl `Template(_row_template).substitute(params)` resultierende Zeichenkette wird anschließend an das bisherige Ergebnis angehängt. Das Beispiel eines Rückgabewerts der Funktion ist in Abbildung 3.17 zu sehen. Die von den Funktionen

```
<div class="LCRow">
[...]
</div>
<div class="LCRow">
    <span class="LCTitle">
        <a href="http://www.studentenwerk-freiburg.de/index.php"
           title="Studentenwerk">Studentenwerk</a>
    </span>
    <div class="discreet">[ Relevanz: 95% ]</div>
    <div class="LCDescr">
        ... Die Seiten des <span class="hl-0">Studentenwerk</span>s
            Freiburg.
    </div>
    <div class="LCUrl">
        <a href="http://www.studentenwerk-freiburg.de/index.php"
           title="http://www.studentenwerk-freiburg.de/index.php">
            <span class="LCUrlHl">www.studentenwerk.uni-freiburg.de
               </span>/index.php</a>
    </div>
</div>
<div class="LCRow">
[...]
</div>
```

Abb. 3.17.: HTML-Fragment der Rückgabe von `getRowsHTML()`.

`getCategoriesMenuHTML()` und `getLanguageMenuHTML()` zurückgegebenen HTML-Fragmente werden auf analoge Weise durch Iterieren über `response.completions` und das Füllen der entsprechenden Templates konstruiert. Dabei werden zusätzlich durch CSS-Anweisungen versteckte HTML-Elemente mit Parametern für das JavaScript der Benutzerschnittstelle eingefügt. So wird beispielsweise dem Eintrag für Deutsch (*de*) im Auswahlmenü für die zu durchsuchende Sprache ein verstecktes `<span>`-Element wie in Abbildung 3.18 angehängt. Dadurch werden die Parameter für die Suchanfrage bei Klicken auf die Sprache festgelegt. Auch der HTML-Code der

[89] Eine Python-Variable des Typs `dict`.

Seitennavigation[90] wird durch versteckte HTML-Elemente ergänzt, die das JavaScript der Benutzerschnittstelle anweisen, welche Suchanfrage bei Klick auf eine Seitenzahl abzusenden ist. Die versteckten Elemente werden nach erfolgreichem Übernehmen der Information durch das JavaScript aus dem HTML-Baum entfernt.[91]

```
<span class="LCParaminfo">q=&amp;language_filter=__:language:de:de&
↪   amp;category_filter=&amp;limit=0&amp;first_call=1</span>
```

Abb. 3.18.: Verstecktes HTML-Element mit Parametern für den Eintrag *de* im Ausklappmenü für die zu durchsuchende Sprache.

3.6. Benutzerschnittstelle

Nachdem die Daten erfasst und geparst, das Backend mit dem erzeugten Index gestartet und der Anfrage-Webserver installiert ist, kann die *Benutzerschnittstelle* (oder auch *Suchseite*) genutzt werden, um Suchanfragen zu stellen und in den Ergebnissen zu navigieren. Siehe hierzu die vorangehenden Abschnitte dieses Kapitels und die Abbildung 3.3.

Im Folgenden werden Aufbau, Realisierung, Funktionsumfang und Bedienung der Suchseite beschrieben.

3.6.1. Aufbau

Die Benutzerschnittstelle besteht aus einem *Zope Page Template*[92] mit der ID[93] `search`, Grafiken, dem JavaScript `complete.js`, sowie dem *Cascading Style Sheet* (*CSS*) `complete.css`. Weiterhin steht das Python-Skript mit der ID `t`[94], für die Übersetzung der verwendeten Zeichenketten in die Benutzersprache, bereit. Das Template `search` liegt unter `/portal_skins/custom` im Plone-Portal und ist durch Acquisition auf jedem Inhaltstyp verfügbar, sodass durch das Anhängen von `/search` an eine beliebige URL innerhalb des Plone-Portals die Suchseite erreicht werden kann.

[90] Wird innerhalb des Moduls durch die Funktion `getPageNavHTML()` erzeugt.

[91] Ist JavaScript im Browser deaktiviert, wird die Funktionalität der Suche über das Absenden des Suchformulars über GET-Parameter umgesetzt. In diesem Fall verbleiben die versteckten HTML-Elemente im HTML-Baum, sind aber weiterhin aufgrund der CSS-Anweisungen unsichtbar. Siehe hierzu auch den folgenden Abschnitt über die Benutzerschnittstelle.

[92] Siehe auch Abschnitt 3.1.1.

[93] Die ID wird innerhalb der ZODB analog zu einem Dateinamen verwendet.

[94] Hier wurde ein sehr kurzer Name gewählt, der die Lesbarkeit des Templates deutlich erhöht.

Alle anderen zugehörigen Objekte liegen unter `/portal_skins/custom/csearch` um die Übersichtlichkeit zu verbessern.

Das Template verwendet *TAL*, *TALES* und *METAL*[95] sowie Python, um während einem ersten Aufruf die HTML-Struktur zu erzeugen. Im HTML-Wurzelelement wird das METAL-Makro `master` wie folgt eingebunden.

```
<html [...]
    metal:use-macro="here/main_template/macros/master"
    [...]>
```

Das Makro definiert *Slots*, die entweder mit Inhalt gefüllt, oder an nachfolgend aufrufende Templates weitergereicht werden können. Die Suchseite füllt im `head`-Element die Slots `javascript_head_slot` und `css_slot`.

```
<head>
    [...]
    <metal:block fill-slot="javascript_head_slot">
        <script type="text/javascript"
            src="csearch/complete.js"></script>
    </metal_block>
    <metal:block fill-slot="css_slot">
        <link rel="stylesheet" type="text/css"
            href="csearch/complete.css"/>
    </metal:block>
</head>
```

Im `body`-Element wird der Slot `main` gefüllt, welcher im Plone-Thema[96] der Universität den Inhaltsbereich darstellt.

```
<div metal:fill-slot="main">
```

Innerhalb des Inhaltsbereichs wird neben dem Seitentitel *Suche* rechtsbündig die Information über die Anzahl der erfassten Dokumente ausgegeben, siehe Abbildung 3.19. Dies geschieht durch die folgenden Anweisungen sowie einige CSS-Anweisungen in `complete.css`.

```
<span tal:replace="
    python:context.csearch.t('a_total_of', 'A total of')" />
<span tal:replace="
    python:context.csearch.get_number_of_documents()" />
<span tal:replace="
    python:context.csearch.t('documents_in_the_index',
        'documents in the index')" />.
```

[95] Siehe ebenfalls Abschnitt 3.1.1.

[96] Sogenannte *Themes* werden in Plone verwendet um das Aussehen und die Funktionsweise der Webseite auf gemeinsame Richtlinien zur Gestaltung, zum Beispiel nach einem Corporate Design, zu vereinheitlichen.

Die beiden Textteile der Nachricht werden durch den Aufruf des Skripts `t` in die aktuell vom Benutzer verwendete Sprache übersetzt. Das Attribut `tal:replace` führt bei Auswertung durch die Template-Engine zum Ersetzen des `span`-Tags durch das Resultat des angegebenen Ausdrucks. Der Präfix `python:` vor dem Ausdruck bewirkt, dass dieser, statt durch die Template-Engine, durch den Python-Interpreter ausgewertet wird. Das zweite `span`-Element wird durch den Rückgabewert des Skripts `get_number_of_documents` ersetzt, welcher – naheliegend – die Anzahl der Dokumente im Index ist.

Die beiden Menüs für die Auswahl der Kategorie und der Sprache, werden initial wie folgt erzeugt:

```
<div id="LCCategories">
    <span id="LCCategoryLabel">
      <span tal:omit-tag=""
          tal:content="python:context.csearch.t(
              'label_category', 'Category')">Category
      </span>:
    </span>
    <span id="LCCategoriesMenuContainer" tal:content="structure
        python:context.csearch.get_categories_menu_html()"/>
</div>
<div id="LCLanguages">
    <span id="LCLanguageLabel">
      <span tal:omit-tag=""
          i18n:translate="label_language">Language
      </span>:
    </span>
    <span id="LCLanguageMenuContainer" tal:content="structure
        python:context.csearch.get_language_menu_html()"/>
</div>
```

Die Beschriftungen werden wieder durch den Aufruf von `t` übersetzt, jedoch werden diesmal die umgebenden `span`-Elemente anschließend entfernt, sodass der übersetzte Text alleine steht. Dies wird durch die Angabe `tal:omit-tag=""` erreicht. Die Einträge der Menüs und deren Trefferzahl, werden durch die Aufrufe der Funktionen `get_categories_menu_html` und `get_language_menu_html` erzeugt. Da die Funktionen HTML zurückgeben, muss innerhalb des `tal:content` als erstes das Schlüsselwort `structure` stehen. Andernfalls würde das HTML entsprechend für die Anzeige umgeschrieben. Die Ausklapp-Funktion der Menüs wird per CSS-Anweisungen in `complete.css` realisiert - das Setzen des gewählten Eintrages wird mit JavaScript erreicht.

Abbildung 3.19 zeigt das aufgerufene Template vor jeglicher Benutzereingabe. Für den Benutzer sichtbar ist lediglich das Suchfeld sowie ein Auswahlmenü für die Kategorie und eines für die Sprache, in der gesucht werden soll. Weiterhin gibt

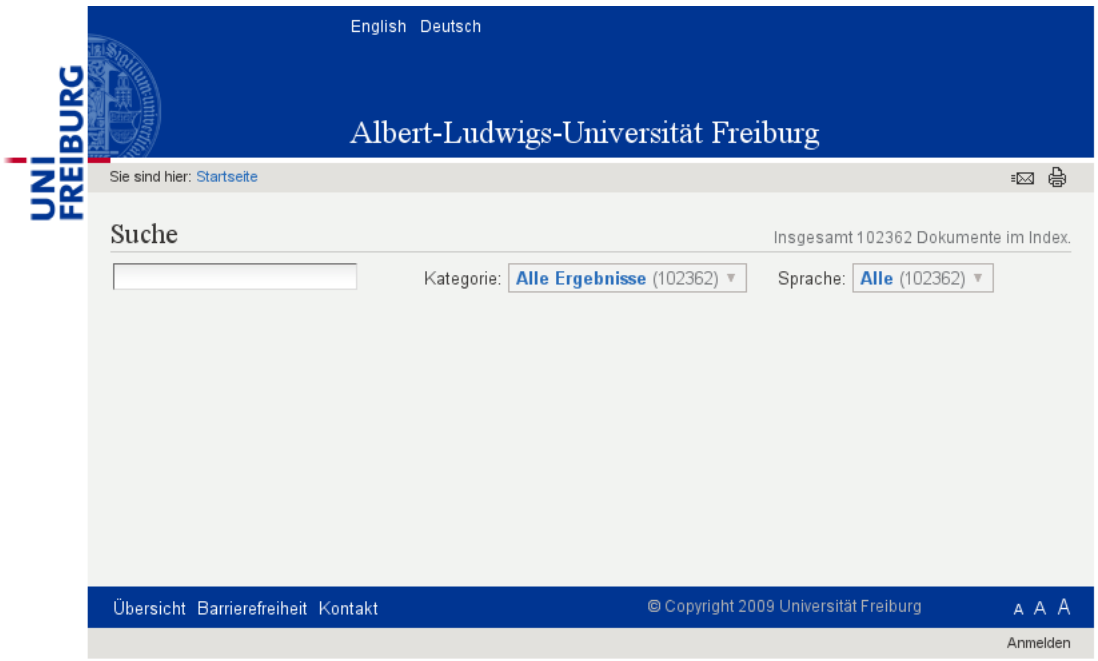

Abb. 3.19.: Die Seite der neuen Suchfunktion nach dem ersten Aufruf. (Links wurde die Navigation der Webseite aus Platzgründen abgeschnitten.)

es eine Schaltfläche mit dem sich die Suche zurücksetzen lässt. Eine Schaltfläche zum Absenden der Suchanfrage fehlt jedoch, da alle Suchanfragen im Hintergrund abgesetzt[97] werden sobald der Benutzer etwas eingibt.[98]

3.6.2. Sofort- und Facettensuche

Ist die Seite erzeugt, wird die gesamte Funktionalität durch das eingebundene JavaScript `complete.js` bereitgestellt.

Das Suchfeld bzw. die Suchfelder[99] werden nach dem Laden der HTML-Struktur mittels jQuery-Aufrufen aus dem eingebundenen `complete.js` um die Funktion der **Sofortsuche** erweitert. Dafür werden alle Suchfelder auf der aktuellen HTML-Seite über deren CSS-Klasse `input.complete-gadget` bzw. deren ID `completeGadget` gefunden und eine JavaScript-Funktion zum Behandeln des Ereignisses `keyup`[100] registriert. Dieser Eventhandler sorgt für das Absenden der Suchanfrage, sobald im

[97] Per *Ajax*: ***A**synchronous **J**avaScript **a**nd **X**ML*.

[98] Nur wenn JavaScript deaktiviert ist erscheint eine solche Schaltfläche, damit auch dann ein Durchsuchen der Seiten möglich ist.

[99] Es reicht nicht, nur ein bestimmtes Suchfeld mit dieser Funktion auszustatten, da eventuell Portlets für den aktuellen Kontext im Plone-Portal definiert sein können und diese ebenfalls ein Suchfeld enthalten können.

[100] Dieses Ereignis wird für das fokussierte Element beim Loslassen einer Taste auf der Tastatur ausgelöst.

Suchfeld mehr als zwei Buchstaben je Wort stehen und der Benutzer für mindestens 400 Millisekunden keine weitere Eingabe getätigt hat. Die Antwort des Anfrage-Webservers wird an die vorgesehene Stelle im HTML-Baum der Seite eingefügt. Damit Suchanfragen nicht doppelt an das Backend gesendet werden müssen, werden die Antworten für alle vom Benutzer eingegebenen Anfragen gespeichert und bei erneuter Eingabe der selben Anfrage aus dem *Zwischenspeicher* geladen. Dies geschieht beispielsweise dann, wenn der Benutzer die Seitennavigation benutzt und dabei zu einer vorher bereits gewählten Seite in den Ergebnissen zurück wechselt. Gleiches gilt für das Navigieren im Browserverlauf, der durch ein JavaScript bei jeder neuen Suchanfrage aktualisiert wird.

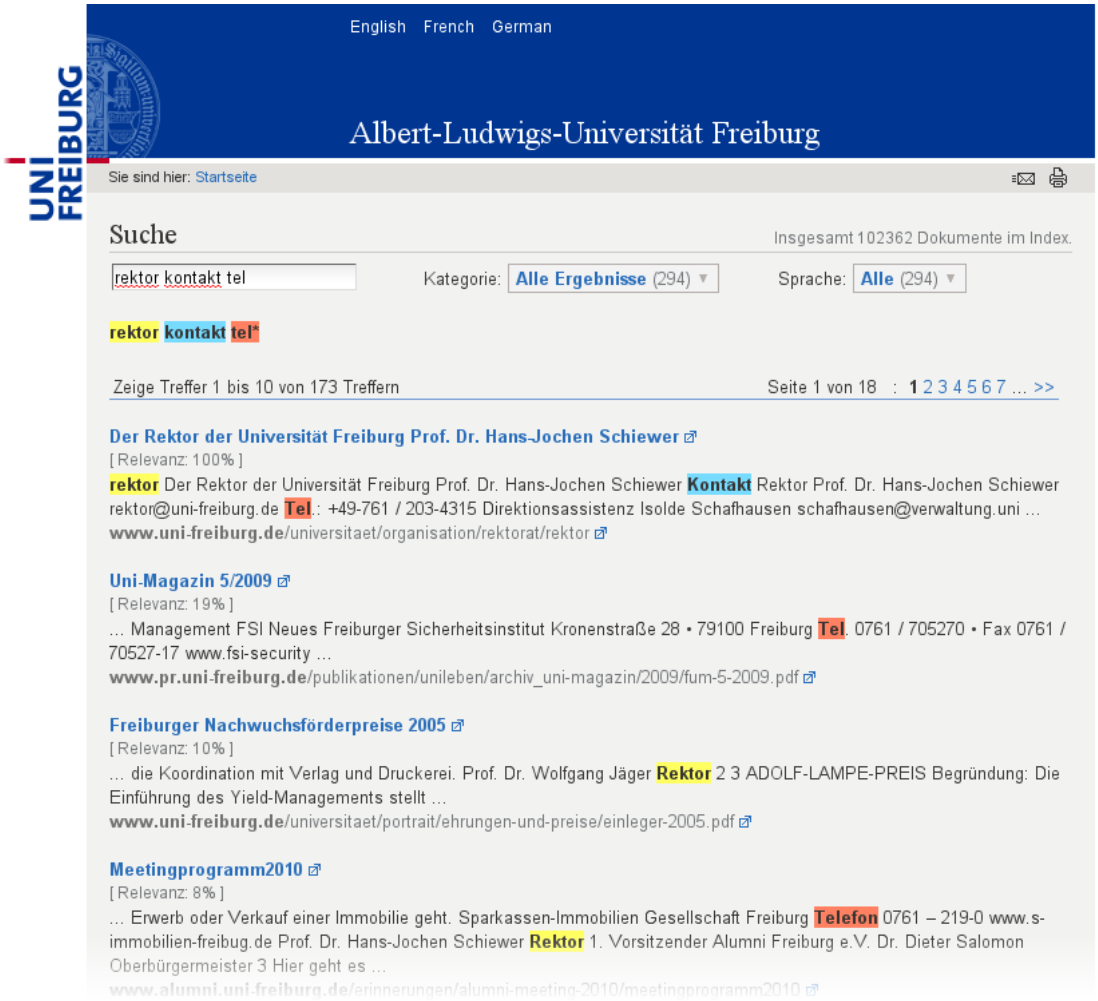

Abb. 3.20.: Suchseite nach erfolgreicher Anfrage `rektor kontakt tel*`.

Jeweils zehn Suchergebnisse werden nach Relevanz absteigend sortiert auf der Seite angezeigt. Weitere Ergebnisse sind durch das Anklicken der Seitennavigation zu erreichen, wobei auch hier, wie schon bei der initialen Suchanfrage, die nächsten

zehn Ergebnisse asynchron im Hintergrund geladen werden. Eine animierte Grafik im Suchfeld zeigt an, dass eine Anfrage abgesendet wurde. Diese wird wieder ausgeblendet, sobald die Ergebnisse sichtbar sind. Abbildung 3.20 zeigt die Suchseite nach erfolgreicher Anzeige der Suchergebnisse für die Anfrage `rektor kontakt tel*`.

Während der Eingabe der Suchanfrage werden dem Benutzer laufend diejenigen Vervollständigungen des letzten Wortes in einem Ausklappmenü direkt unterhalb des Suchfeldes angeboten, welche die höchste Gewichtung erhalten haben. Dabei wird auch hier die Anzahl der Treffer hinter jeder Vervollständigung angezeigt. Die gewünschte Vervollständigung kann einfach per Mausklick oder per Tastatur mit den Pfeiltasten und der Eingabetaste ausgewählt werden. Abbildung 3.21 zeigt das beschriebene Ausklappmenü.

Abb. 3.21.: Ausklappmenü der Auto-Suggest-Funktion der neuen Suchseite.

Ebenso wie die Vervollständigungen werden auch die Menüs für die Einschränkung der Sprache und der Kategorien bei jedem Tastenanschlag aktualisiert. Es werden dabei nur die Sprachen und Kategorien angezeigt, welche für die aktuelle Suchanfrage zu Treffern führen. Die Treffer sind in Klammern hinter den Einträgen der Menüs vermerkt. Die so realisierte *Facettensuche* erlaubt Entscheidungen über die zu durchsuchende Kategorie oder Sprache dann zu treffen, wenn sie intuitiv notwendig erscheint, wobei die Information über die resultierenden Trefferanzahlen im Vorhinein gut sichtbar ist und somit den Suchprozess erleichtert.

3.6.3. Benutzersprache

Plone erlaubt es dem Benutzer seine bevorzugte Sprache zu wählen. Dies geschieht weitgehend automatisch durch den Internetbrowser oder manuell über einen Link. Alle Navigationselemente sind dann in der jeweiligen Sprache verfügbar. Artikel innerhalb des CMS können in alle verfügbaren Sprachen übersetzt werden und erscheinen dann nur, wenn der Benutzer diese Sprache gewählt hat. Auch die neue Suchseite

ist in derzeit zwei Sprachen – Deutsch und Englisch – verfügbar. Allerdings wird die englische Version als Standard verwendet, falls keine Übersetzung der Suchseite in der aktuellen Benutzersprache verfügbar ist. Abbildung 3.22 zeigt die englische Version in Aktion.

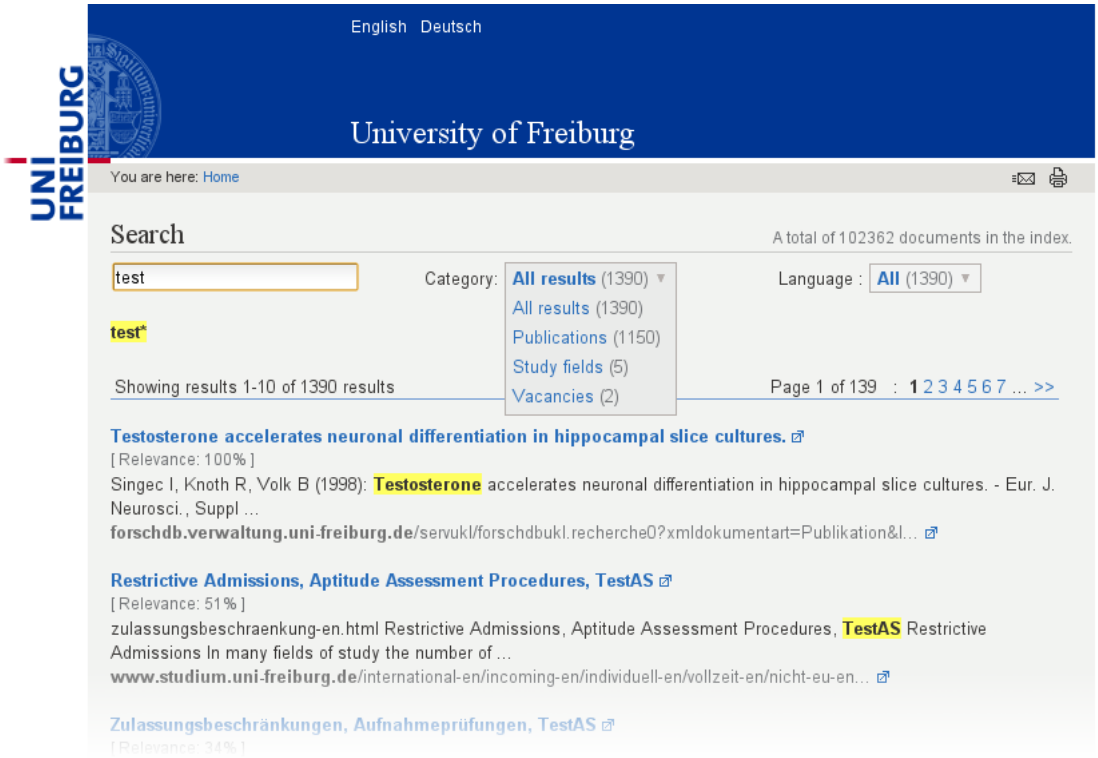

Abb. 3.22.: Englische Version der Suchseite.

Das Übersetzen erfolgt durch das oben bereits erwähnte Skript `t`. Das Skript `t` akzeptiert die zwei Parameter `msgid` und `default`. Der erstere ist die eindeutige ID der zu übersetzenden Zeichenkette, im zweiten Parameter kann ein Wert angegeben werden, der zurückgegeben wird, falls keine Übersetzung existiert.

Als ersten Schritt, ruft das Skript die Externe Methode `getCurrentLanguage()` aus `complete.py` auf, welche die zweistellige Länderkennung der aktuell eingestellten Benutzersprache zurückgibt, und speichert diesen in der Variablen `lang`. Das Skript enthält eine Python-Variable vom Typ `dict`, welche die zweistellige Länderkennung wiederum einem weiteren Wörterbuch zuordnet, welches eine ID der Übersetzung zuordnet, siehe Abbildung 3.23.

In einem zweiten und dritten Schritt wird durch das aufeinander folgende Nachschlagen von `lang` in `translations` und von `msgid` in `translations[lang]` die Übersetzung gefunden und dann zurückgegeben. Fehlt die Sprache oder die ID der Zeichenkette im Wörterbuch, wird `default` zurückgegeben. Englisch ist die Standardsprache und über den Parameter `default` des Skripts sind alle Zeichenketten

```
translations = {
    u'de': {
        u'a_total_of': u'insgesamt',
        u'documents_in_the_index': u'Dokumente im Index',
        u'all_languages': u'alle',
        u'all_results': u'Alles',
        u'showing_from_to_of':
            (u'Zeige Treffer ${show_from} bis ${to}'
             u' von ${total} Treffern'),
        u'to_first_page': u'Zur ersten Seite',
        u'to_page': u'Zur Seite ${page_number}',
        u'to_last_page': u'Zur letzten Seite',
        u'page_x_of_y': u'Seite ${x} von ${y}',
        [...]
        }
    }
```

Abb. 3.23.: Die Variable `translations` im Skript `t` (gekürzt).

an der entsprechenden Stelle im Quelltext auf Englisch vorgegeben. Daher ist kein Eintrag für `en` in `translations` zu finden.

Auch der Anfrage-Webserver verwendet in `complete.py` das Skript `t` um Zeichenketten zu übersetzen. Dieser Ansatz ist gewählt worden, um die Übersetzungen über das ZMI für den Verantwortlichen Administrator des Plone-Portals leicht zugänglich an *einer* Stelle zu sammeln. Würden die Übersetzungen der Zeichenketten in `complete.py` direkt in der Datei übersetzt, müsste bei jeder Änderung der Übersetzung die Datei editiert und anschließend die Zope-Instanz neu gestartet werden. Dafür bräuchte der Plone-Administrator zugriff auf das Dateisystem des Servers, was aus Sicherheitsgründen am Rechenzentrum nicht zugelassen wird.

3.6.4. Browserverlauf und Lesezeichen

Für jede erfolgreich gestellte Suchanfrage, dem Wählen einer Kategorie oder Sprache, oder dem Navigieren in den Suchergebnissen, wird der Verlauf des Internetbrowsers über `complete.js` manipuliert. Falls kein *Fragmentbezeichner*[101] (`#`) in der URL des Browsers vorhanden ist, wird er eingefügt und die Parameter der Suchanfrage kodiert angehängt. Die zweite Seite der Ergebnisse auf die Suchanfrage `test` in der Sprache *Englisch* und der Kategorie *Publikationen* wird dann etwa so kodiert:

```
#q=test&l=en&c=publications&p=2
```

[101] Englisch: *fragment identifier*.

Durch das Ändern der URL in der Adresszeile des Browsers wird ein Eintrag in dessen *Verlauf* erzeugt und der Benutzer kann mit den gewohnten Kommandos zwischen mehreren Ergebnissen vor- und zurückwechseln, ohne die Suchanfrage erneut eingeben zu müssen.

Da die URL in der Adresszeile manipuliert wird, lässt sich wie gewohnt ein Lesezeichen erstellen, welches direkt auf das Suchergebnis verweist. Wird eine solche URL aufgerufen, füllt `complete.js` nach dem Laden der Suchseite das Suchfeld mit der Suchanfrage und löst das Ereignis `keyup` aus, sodass die Suchanfrage ohne Zutun des Benutzers gestellt wird. So ist es auch möglich durch die URL auf ein bestimmtes Suchergebnis hinzuweisen, einen Link auf einer anderen Seite zu setzen, oder in einer E-Mail zu referenzieren.

Der Microsoft Internet Explorer (IE) erzeugt leider erst ab der Version 8 einen Eintrag im Browserverlauf wenn sich der Fragmentbezeichner ändert, sodass für frühere Versionen ein sogenannter Eingebetteter Frame (`iframe`) versteckt eingefügt werden muss. Dessen Inhalt lässt sich dann manipulieren und sich dadurch ein Eintrag im Browserverlauf erzeugen.

3.7. Performance und Optimierung

Die entwickelte neue Suchfunktion erlaubt das Suchen in den erfassten Daten in Lokalität und Beschaffenheit verschiedener Quellen über eine Sofortsuche, die unter Verwendung eines leistungsschwachen Testservers[102] für das Plone-Portal und einem handelsüblichen Notebook[103] im Durchschnitt *etwa 360 Millisekunden* benötigt um Ergebnisse präsentieren zu können. Im Folgenden soll ein kurzer Überblick über die Performance und den Hauptspeicherbedarf der Komponenten gegeben sowie die getroffenen Maßnahmen zur Optimierung aufgezeigt werden.

3.7.1. Datenerfassung

Die *Gesamtlaufzeit*[104] einer ersten Aktualisierung mit den derzeit indexierten Quellen[105] beträgt *etwa 45 Minuten*. Jede weitere Aktualisierung dauert etwa 25 Minuten. Davon entfallen auf die Plone-Portale etwa 23 Minuten für eine Erstaktualisierung und eine Minute für eine Folgeaktualisierung. Der Zeitaufwand für jede Aktualisierung von nicht-Plone-Quellen beträgt also etwa 20 Minuten, da keine Konsolidierung

[102] AMD Athlon(tm) 64 X2 Dual Core Processor 4600+, 512 Kilobyte Cache, Ubuntu 8.04 Hardy Heron, 4 Gigabyte Hauptspeicher, SATA-Festplatte.

[103] Intel(R) Core(TM)2 Duo CPU T5870 @ 2.00GHz, 2 Megabyte Cache, Ubuntu 10.04 Lucid Lynx, 3 Gigabyte Hauptspeicher. Browser: *Chromium* v. 7.0.510.0 und *Firefox* v. 3.6.8.

[104] Unter der verwendeten Hardware – siehe 3.2.

[105] Siehe Tabelle 3.1.

der vorhandenen mit den neuen Daten möglich ist, und so immer alle Daten der Quellen angefordert werden müssen. Das Skript `removeInvalidUTF8` verarbeitet etwa 1,8 Megabyte Daten pro Sekunde und hat so bei derzeit 90 Megabyte Rohdaten einen Anteil von etwa 50 Sekunden an der Gesamtlaufzeit.

Die Datenerfassung wurde dahingehend optimiert, dass für eine Aktualisierung des Index möglichst wenig Daten über das Netzwerk transportiert werden müssen. Gleichzeitig wird die Last und der Hauptspeicherbedarf des Rechners zur Datenerfassung niedrig gehalten.

Plone-Portale: Zusätzlich zu den oben genannten Punkten, wurden für Plone-Portale zudem Optimierungen durchgeführt, um den Hauptspeicherbedarf des Zope-Servers der zu indexierenden Portale gering zu halten. Dies wird durch das paketweise, inkrementelle Aktualisieren der Portaldaten über `remoteSyncQueryXML` erreicht.

Auf dem Zope-Server des abgefragten Plone-Portals müssen die Objekte, der seit der letzten Aktualisierung geänderten Einträge, in den Hauptspeicher des Servers geladen werden, da der Volltext der Einträge durch das Aufrufen der Funktion `SearchableText()` direkt auf den Objekten gewonnen werden muss. Dieses in den Hauptspeicher laden, oder *Aufwecken*, der Objekte in der ZODB kostet sowohl *Hauptspeicher* des Servers als auch *Zeit*. Durch das Einführen eines Parameters `last_query` im Skript `remoteSyncQueryXML` müssen jedoch nur die seit dem angegebenen Datum geänderten Objekte in den Speicher geladen werden. Für nicht modifizierte Objekte wird nur die URL des jeweiligen Eintrags innerhalb des Plone-Portals aus dem Index des `portal_catalog` benötigt, wodurch ein Aufwecken der Objekte nicht notwendig ist und die zu übertragenden Datenmenge auf die URL des Eintrags beschränkt bleibt[106].

Tabelle 3.4 zeigt, dass sich durch den Einsatz des Parameters `last_query` die *Laufzeit* der Abfrage der Daten für die Portale *im Durchschnitt auf 0,86% reduziert*, wenn sich *keine Einträge der Portale geändert*[107] haben, also alle Objekte allein mit den Daten des `portal_catalog` repräsentiert werden können.

In einem weiteren Test zeigt sich, dass sich der Zeitaufwand für eine Aktualisierung eines Portals mit *20 modifizierten Einträgen* nur bei *4,8% der Zeit* einer Erstaktualisierung liegt. Siehe Tabelle 3.5.

Um den *Hauptspeicherbedarf* des Zope-Servers des abgefragten Portals und des Rechners für die Datenerfassung weiter zu minimieren, werden die Parameter `increment` und `from_idx` des Skripts `remoteSyncQueryXML` eingesetzt. Ersterer gibt an, wie viele Einträge gesendet werden sollen, letzterer wie viele Einträge bereits

[106] Zuzüglich einer konstanten Anzahl an Bytes für das mitgesendete Markup.

[107] Ausgenommen das Studierendenportal, für welches über `remoteSyncQueryXMLAdditional` eine MySQL-Datenbank abgefragt wird, die kein Modifikationsdatum speichert.

Portal	A $[s]$	A_m	A_u	B $[s]$	B_m	B_u	$\frac{B}{A}$
`studium`*	353,16	3619	0	6,61	181	3438	0,0187
`uni`**	446,61	1622	0	1,82	0	1622	0,0041
`alumni`	217,68	1206	0	2,23	0	1206	0,0103
`mw`	128,31	1152	0	0,47	0	1152	0,0037
`podcasts`	46,72	1109	0	0,50	0	1109	0,0107
`pr`	30,36	344	0	0,22	0	344	0,0073
`exzellenz`	22,61	179	0	0,13	0	179	0,0058

Tab. 3.4.: Effekt des Parameters `last_query` auf die Laufzeit der Datenerfassung für Portale, ohne Konsolidierung. Messungen durchgeführt am 12.07.2010. Alle Daten wurden in einem HTTP-Request abgerufen, d.h. der Parameter `increment` wurde auf 10000 gesetzt. Spalte A enthält die Werte für `last_query=1970/01/01`, Spalte B für `last_query=2010/07/11`. A_m, B_m sind jeweils die Anzahl modifizierter Einträge, A_u, B_u die unmodifizierter. *Das Portal `studium` sendet 181 Einträge aus der Abfrage einer MySQL-Datenbank über `remoteSyncQueryXMLAdditional`. **Das Portal `uni` ist eine Kopie zu Test- und Entwicklungszwecken.

Portal	A $[s]$	B $[s]$	C $[s]$	D $[s]$	$\frac{D}{A}$
`uni`*	446,75	3,16	11,54	21,43	0,0480

Tab. 3.5.: Effekt des Parameters `last_query` auf die Laufzeit der Datenerfassung für Portale, ohne Konsolidierung (2). Messungen durchgeführt am 17.07.2010. Alle Daten wurden in einem HTTP-Request abgerufen, d.h. der Parameter `increment` wurde auf 10000 gesetzt. Spalte A enthält die Werte für `last_query=1970/01/01`. Spalte B enthält die Zeit für eine Aktualisierung mit 2 modifizierten Einträgen, Spalte C für 10 modifizierte Einträge und D für 20 modifizierte Einträge. *Das Portal `uni` ist eine Kopie zu Test- und Entwicklungszwecken.

empfangen wurden. Der Standardwert für `increment` ist auf `3` festgelegt, wodurch für jede HTTP-Anfrage nur drei Einträge des Portals angefragt werden. Über den Parameter `from_idx` wird das Skript `remoteSyncQueryXML` angewiesen nur Einträge ab diesem Index zu senden.

Die insgesamt *9231 Einträge der Portale* der indexierten Portale haben eine *durchschnittliche Größe von 1,88 Kilobyte*. So müssen bei `increment=3` auf Seiten des Rechners für die Datenerfassung im Durchschnitt nur 5,64 Kilobyte in den Hauptspeicher geladen werden. Dies ist ein sehr kleiner Wert. Potentiell sind jedoch Einträge von hundert Megabyte Größe oder mehr denkbar, von denen dann im schlimmsten Falle drei gleichzeitig in den Hauptspeicher geladen werden müssten.

Durch das Abfragen der Portaldaten in Paketen zu drei Einträgen wird als Kompromiss der Overhead für die Netzwerkkommunikation erhöht. Da der Rechner für die Datenerfassung, stromboli[108] eine schnelle Anbindung innerhalb des Uni-Netzes hat, hält sich der zeitliche Overhead jedoch in *derzeit vertretbaren* Grenzen. So reduziert sich die Gesamtlaufzeit einer Erstaktualisierung lediglich von 45 auf 40 Minuten, wenn der Wert des Parameters `increment` von drei auf zehn erhöht wird. Da jedoch der Anteil der Plone-Portale bei 23 Minuten liegt, bedeutet die Erhöhung des Parameters eine Erhöhung des zeitlichen Overheads von etwa 22%. Sollte nach dem Hinzufügen vieler Plone-Portale in der Zukunft zeitlicher Optimierungsbedarf bestehen, ist eine erneute genauere Analyse der Größe der Einträge erforderlich, sodass gegebenenfalls `increment` erhöht werden kann.

Durch effizientes Konsolidieren der neuen mit den vorhandenen Daten durch die Funktion `_consolidate()` im Python Modul `buildxml.plugins.portal` wird Speicheraufwand und Rechenzeit des Rechners zur Datenerfassung weiter minimiert. Das *Konsolidieren* der Daten ist dahingehend zeitlich optimiert, dass die Zeiger innerhalb der zu konsolidierenden Dateien immer nur weiter in Richtung Dateiende verschoben werden. Dies wird durch die Sortierung der gesendeten Einträge nach Modifikationsdatum möglich. Durch diese Art der Positionierungen[109] der Dateizeiger können die Zugriffe auf die Festplatte zeitlich optimal durchgeführt werden. Da die Inhalte beider Dateien nur eintragsweise gelesen werden, ist der Hauptspeicherverbrauch durch die Größe der einzelnen Einträge beschränkt.

Da derzeit die Datenerfassung lediglich *zwei mal am Tag* gestartet wird, ist die *Gesamtlaufzeit* der Aktualisierung mit etwa 45 Minuten *nicht kritisch* für den reibungslosen Betrieb. Erst wenn die Dauer einer Aktualisierung die 12-Stunden-Marke überschreitet, sind Aktualität und Integrität der Daten nicht mehr gewährleistet. Wenn alle neu hinzukommenden Quellen Plone-Portale sind, und der durchschnittli-

[108] Für eine kurze Beschreibung der Hardware siehe Abschnitt 3.2.
[109] Englisch: *to seek*: *positionieren*.

che Umfang bei aufgerundet 1319 Einträgen liegt[110], dauert die Datenerfassung für jede dieser Quellen durchschnittlich *etwa drei Minuten* länger. So können noch mehr als 200 Portale hinzugefügt werden, bevor besagte Grenze erreicht ist.

Forschungsdatenbank: Die Forschungsdatenbank enthält alle eingetragenen *Publikationen* aller Fakultäten, Arbeitsgruppen und Wissenschaftler der Universität und des Universitätsklinikums. Die Abfrage der Daten erfolgt per HTTP-Anfrage an einen Tamino-Server des Rechenzentrums des Universitätsklinikums. Dies wird für jede Fakultät einzeln durchgeführt, um den *Tamino XML-Server*[111] des Rechenzentrums zu entlasten. Die Antwort ist ein XML-Dokument mit allen Forschungsberichten der Fakultät. Es werden dann mittels *BeautifulStoneSoup* die Publikationen extrahiert und entsprechend ihres Typs formatiert als jeweils ein Eintrag gespeichert. Eine Aktualisierung der Daten dauert etwa *12 Minuten*, eine Konsolidierung findet nicht statt, da kein Modifikationsdatum in der Datenbank gespeichert ist. In Tabelle 3.6 sind die Werte der Zeitmessung einer Aktualisierung für das Plugin `forschdb` zu sehen. Die Datenmenge der XML-Antworten übersteigt *52 Megabyte* nicht, kann also problemlos in den Hauptspeicher geladen werden.

Studentenwerk: Die ca. 500 Einträge des Studentenwerks werden innerhalb von *etwa 10 bis 15 Minuten aktualisiert*. Leider verwendet der Typo3-Server des Studentenwerks nur bei fünf der derzeit indexierten HTML-Seiten das Header-Feld `Last-Modified` - die Header-Felder `ETag` oder `MD5` werden überhaupt nicht verwendet, weshalb sich eine Konsolidierung mit vorhandenen Daten nicht spürbar zeitsparend auswirken würde.

Stellenbörse und Veranstaltungskalender: Sowohl die Einträge der Stellenbörse, als auch die des Veranstaltungskalenders, sind innerhalb von *1 bis 2 Sekunden* erfasst. Für beide Quellen werden *alle Einträge auf* einmal abgefragt, da jeder Eintrag in der Regel unter einem Kilobyte groß ist.

3.7.2. Indexierung

Um die Indexdatei `unifr.hybrid` aus der 90 Megabyte großen Datei mit den zu indexierenden Einträgen `unifr.xml` zu erzeugen, dauern das Parsen, Sortieren der Wörter und Erzeugen der Datenstruktur HYB *insgesamt etwa 32 Sekunden.*[112]

[110] Arithmetisches Mittel der Anzahl der Einträge der am 11.07.2010 indexierten Portale.

[111] Ein Tamino-Server stellt XML-Dokumente als Datenbank zur Verfügung. Tamino wird von der *Software AG* (www.softwareag.com) entwickelt.

[112] Alle Messungen zur Indexierung wurden auf `stromboli` durchgeführt. Für eine genauere Beschreibung des Systems siehe 3.2.

Fakultät oder Einrichtung	Antwort [s]	Lesen [s]	BSS [s]	#Einträge
Universitätsklinikum	14,40	131,85	190,43	50539
Zentrale Einrichtungen	0,01	0,00	0,00	0
Theologische F.	0,22	1,52	12,33	1693
Rechtswissenschaftliche F.	0,46	1,68	9,60	2320
Wirtschafts- u. Verhaltenswiss. F.	0,63	2,52	7,84	3270
Medizinische F. (Vorklinik)	1,67	4,71	12,58	4355
Philologische F.	0,45	4,03	13,23	4986
Philosophische F.	0,40	3,13	8,70	3656
F. für Mathematik u. Physik	1,20	3,34	7,64	3439
F. für Chemie, Pharmazie, Geowiss.	1,34	3,57	9,00	3451
F. für Biologie	0,71	1,95	4,19	1740
F. für Forst u. Umweltwiss.	1,30	4,28	12,44	5191
Technische Fakultät	0,90	3,52	9,72	4047

Tab. 3.6.: Zeitmessungen für das Plugin `forschdb`. Die Spalte *Antwort* enthält die vom Tamino-Server benötigte Zeit um auf die Anfrage zu antworten. Die Spalte *Lesen* enthält die benötigte Zeit für das Empfangen der vollständigen Antwort. Die Spalte *BSS* enthält die Zeit, die BeautifulStoneSoup benötigt um die Datenstruktur aus den empfangenen Daten aufzubauen. Die Gesamtlaufzeit des Plugins war *11 Minuten und 26 Sekunden. 8 Minuten und 12 Sekunden* (entspricht *72%*) davon entfallen auf das Senden und Empfangen der Antwort sowie das Aufbauen der BeautifulStoneSoup-Datenstruktur. Das Universitätsklinikum mit ca. 57% des Gesamtvolumens an Einträgen hat auch den größten Anteil an der Gesamtlaufzeit.

Während der *Indexierung* verarbeitet der *Parser* `unifr.xml` innerhalb von *etwa sechs Sekunden*, das entspricht einem *Datendurchsatz von 15 Megabyte pro Sekunde*. Es können also *etwa 0,88 Gigabyte pro Minute* oder 52 Gigabyte Daten pro Stunde verarbeitet werden. Das *Sortieren* der Wörter benötigt *etwa 18 Sekunden*. Das Erzeugen der Indexstruktur HYB erfolgt in *etwa sieben Sekunden*.

3.7.3. Backend

Das Backend beantwortet Anfragen im Durchschnitt innerhalb von *etwa 9 Millisekunden*. Dies ist das arithmetische Mittel der Werte des Testservers auf `stromboli`[113]. Es mussten keine weiteren Optimierungen durchgeführt werden.

3.7.4. Anfrage-Webserver

Die vom Anfrage-Webserver benötigte Zeit für das Absenden, Empfangen und Verarbeiten einer Suchanfrage liegt im Mittel bei 32,33 Millisekunden[114].

Der Anfrage-Webserver wurde dahingehend optimiert, dass eine möglichst schnelle Verarbeitung der Antwortdaten des Backends und Erzeugung der HTML-Fragmente stattfindet. Für das Verarbeiten der XML-Antwort des Backends wird der Parser `xml.parsers.expat` eingesetzt, welcher eine hohe Performance aufweist. Zudem werden einfache Richtlinien für das Schreiben effizienter Python-Programme berücksichtigt. So werden häufige Lookups in den internen lokalen und globalen Variablentabellen des Python-Interpreters minimiert[115], die Klasse `string.Template` beziehungsweise der Operator `%` für Einfügen und Anhängen von Zeichenketten verwendet. Es wird weiter berücksichtigt, dass Zeichenketten in Python unveränderlich sind. Daher werden Zeichenketten-Manipulationen wo möglich und sinnvoll[116] zusammengefasst, um den Overhead für das Erzeugen neuer String-Objekte gering zu halten. Wenn vorhanden, wurden Python-eigene Funktionen selbst geschriebenen vorgezogen, da die eingebauten Funktionen in `C` geschrieben und damit sehr viel performanter sind. Für einen Überblick über mögliche Optimierungen von Python-Programmen, siehe [7].

[113] Siehe Abschnitt 3.2.
[114] Gemessen auf dem Testserver `cmsdev.rektorat.uni-freiburg.de` mit einem AMD Athlon(tm) 64 X2 Dual Core Prozessor 4600+ und 4 Gigabyte Hauptspeicher unter Ubuntu 8.04.
[115] Vermeiden des Punkt-Operators (`.`) zum Ansprechen von Attributen.
[116] Lesbarkeit ist ein weiteres Kriterium.

3.7.5. Benutzerschnittstelle

Das Einfügen der Ergebnisse einer Anfrage und das Hinzufügen der JavaScript-Funktionalität benötigt im arithmetischen Mittel *etwa 50 Millisekunden*[117]. Dabei ist der größte Posten mit etwa 80% das Einfügen der Ergebnisse in die HTML-Struktur der Seite. Das JavaScript wurde durch Ausprobieren, scharfes Überlegen und unter Einbeziehung der gemessenen Zeiten auf eine möglichst kurze Laufzeit optimiert.

3.8. Zwischenfazit

Es wurde ein kurzer Überblick über Zope, Plone, die verwendeten Programmiersprachen und die Architektur der neuen Suchfunktion gegeben. Anschließend wurde die Implementierung der Datenerfassung, des Parsers zur Indexierung, des Anfrage-Webservers und der Benutzerschnittstelle genauer beschrieben. Ebenso wurden die Funktionsweise und Bedienung sowie die verfügbaren und verwendeten Funktionen des Backends erläutert. Im Zusammenspiel erlauben die fünf Komponenten das Erfassen von Informationen aus verschiedenen Quellen und das Durchsuchen dieser mittels einer intuitiven Suchseite mit lediglich einem Suchfeld. Es werden die Paradigmen der Sofort- und Facettensuche für ein effizientes und intuitives Suchen verwendet. Die Ergebnisse werden nach Relevanz gewichtet und mit Auszügen und hervorgehobenen Suchworten versehen angezeigt.

Der Performance wurde durch zahlreiche Optimierungen Rechnung getragen. Die Daten aus elf Quellen mit insgesamt über 90.000 Datensätzen werden innerhalb von derzeit maximal 45 Minuten erfasst, mit einer Datenrate von etwa 15 Megabyte pro Sekunde indexiert und stehen dann für Suchanfragen an das Backend zur Verfügung. Eine solche Suchanfrage durch den Benutzer kann in einem Drittel einer Sekunde beantwortet werden.

Die entwickelten Komponenten der neuen Suchfunktion sind nun einsatzbereit und arbeiten wie in Abbildung 3.3 gezeigt zusammen.

[117] Das Testsystem war hier wieder das Notebook, wie im einleitenden Absatz des Abschnitts 3.7 beschrieben.

4. Administration

In diesem Kapitel sollen die erforderlichen Maßnahmen und Verhaltensweisen für einen reibungslosen Betrieb der neuen Suchfunktion erläutert, sowie Hinweise und Strategien zur Eingrenzung von Fehlerquellen und der Weiterentwicklung des Systems gegeben werden.

Es ist dazu wichtig, das netzwerktechnische Umfeld der Suchfunktion im Blick zu haben, weshalb zuerst der Ablauf einer HTTP-Anfrage an einen Zope-Server des Rechenzentrums der Universität erklärt werden soll. Auftretende Fehler lassen sich oftmals durch die Meldungen in Logdateien oder das Debugging der entsprechenden Komponente nachvollziehen und beheben, weshalb das Logging und die Fehlersuche in den folgenden Abschnitten genauer beschrieben wird. Schließlich wird im letzten Abschnitt die Verfügbarkeit des Quelltextes und seiner Online-Dokumentation erklärt.

4.1. Ablauf von HTTP(S)-Anfragen

Ein Besucher der Webseiten der Universität sendet über seinen Internetbrowser HTTP-Anfragen an Port 80 eines Webservers. Der Webserver[1] sendet ihm die angeforderten Inhalte zurück und der Internetbrowser stellt diese dar.

Eine HTTP-Anfrage an Port 80 passiert die *Firewall* der Universität und wird vom *Router* an einen *Reverse-Proxy* (*Squid*) weitergeleitet. Ist die angeforderte Ressource im *Cache* vorhanden und valide, wird die Antwort direkt aus diesem bezogen. Ist der Inhalte jedoch nicht im Cache oder nicht mehr gültig, wird die Anfrage an einen *Load-Balancer* (*Pound*) weitergeleitet, welcher sie wiederum an einen *ZEO-Client* mit genügend freien Kapazitäten zum Bearbeiten der Anfrage weiterleitet. Der ZEO-Client kommuniziert für die Beantwortung der Anfrage mit dem *ZEO-Server*, welcher wiederum mit der Objektorientierten Datenbank *ZODB* kommuniziert.

Falls über *HTTPS* kommuniziert wird, wie etwa, wenn der Benutzer zur Bearbeitung der Portalinhalte oder für die Administration angemeldet ist, wird der *Cache umgangen* und direkt mit dem Load-Balancer kommuniziert. Dies verhindert Konflikte beim Bearbeiten der CMS-Inhalte aufgrund veralteter Cache-Einträge.

[1] Zope beinhaltet einen eigenen integrierten Webserver *ZServer* auf der Basis des in Python geschriebenen High-Performance-Frameworks für socketbasierte Server *Medusa*. Für weitere Informationen ist der Quelltext von Zope die beste Ausgangsstelle.

Einen Überblick über den Ablauf der HTTP(S)-Anfragen an einen Zope-Server der Universität wird in Abbildung 4.1 gegeben.

Der HTTP-Server nimmt die HTTP-Anfrage entgegen und kapselt diese in einem `HTTPRequest`-Objekt, welches um Informationen zum User-Agent[2] und der Konfiguration des Servers ergänzt wird. Das Anfrage-Objekt wird verarbeitet und die *Z Object Database* ausgehend vom Wurzelelement des Inhaltsbaumes `OFS.Application` unter der Verwendung der Funktionen `traverse()` und `__bobo_traverse__()` *traversiert*, bis das, der URL der Anfrage entsprechende, Objekt gefunden wurde. Das gefundene Objekt wird in den Speicher geladen[3], indem die Funktion `__call__()` des Objekts aufgerufen wird. Diese Funktion fragt im Plone-Tool `portal_types` nach der zu verwendenden *Anzeigemethode* und in `portal_skins` nach dem für die Anzeige zu verwendenden *Template*. Anschließend wird das Template durch das Auswerten der *TAL/TALES*- und *METAL*-Ausdrücke gefüllt und die fertige Seite vom HTTP-Server als *Antwort gesendet*. Abbildung 4.2 zeigt den stark vereinfachten Ablauf einer HTTP-Anfrage und deren Beantwortung durch einen ZEO-Client.

4.2. Verfügbarkeit und Aktualität

Die Aktualität der indexierten Daten ist wichtig für den produktiven Betrieb, um möglichst alle Seiten schnell über die neue Suchfunktion zugänglich zu machen und nicht auf nicht mehr existierende Inhalte zu verlinken. Ebenso sollte die Suchfunktion nicht ausfallen und den Benutzer ohne weitere Suchmöglichkeiten vor einer Seite mit der Fehlermeldung stehen lassen. Daher sind einige Vorkehrungen getroffen worden, um Aktualität und Ausfallsicherheit der Komponenten zu gewährleisten.

Datenerfassung und Indexierung: Das Skript `getXML.py` wird per *Cronjob* (siehe Abbildung 4.3) zweimal am Tag aufgerufen, der Index neu erstellt und anschließend das Backend mit dem neuen Index gestartet. Tritt ein Fehler auf wird eine E-Mail mit zusammenfassenden Informationen an den Administrator versendet. Die alte Version des Index wird dann solange weiter verwendet bis ein fehlerfreier Durchlauf der Datenerfassung erfolgt ist. Ein konsekutiver Lauf des Skripts auf dem Entwicklungssystem[4] dauert derzeit zwischen 25 und 30 Minuten.[5]

Für bestimmte Inhalte ist es wichtig, dass sie schnell in den Suchergebnissen gefunden werden. Pressemitteilungen der Abteilung Presse- und Öffentlichkeitsarbeit und andere – meist kurzlebige – Inhalte sollten schnellstmöglich gefunden werden können.

[2] Normalerweise der Internetbrowser.
[3] Man spricht auch vom *Aufwecken* des Objekts – eine zeitintensive Operation.
[4] Für die Daten der Hardware des Systems siehe Abschnitt 3.2.
[5] Siehe auch Abschnitt 3.7 über Performance und Optimierung.

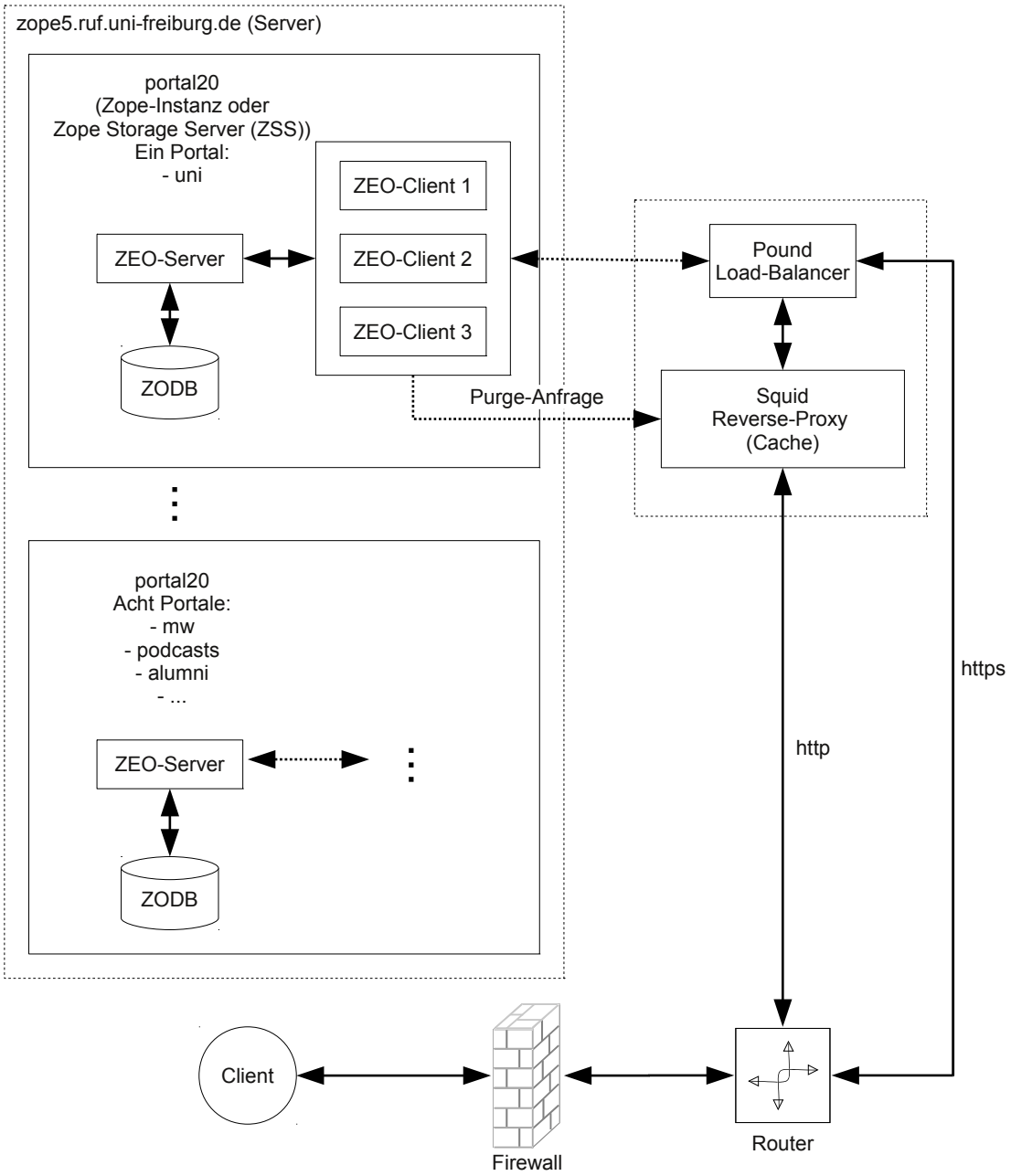

Abb. 4.1.: Ablauf der HTTP-Anfragen an den Zope-Server `zope5`.

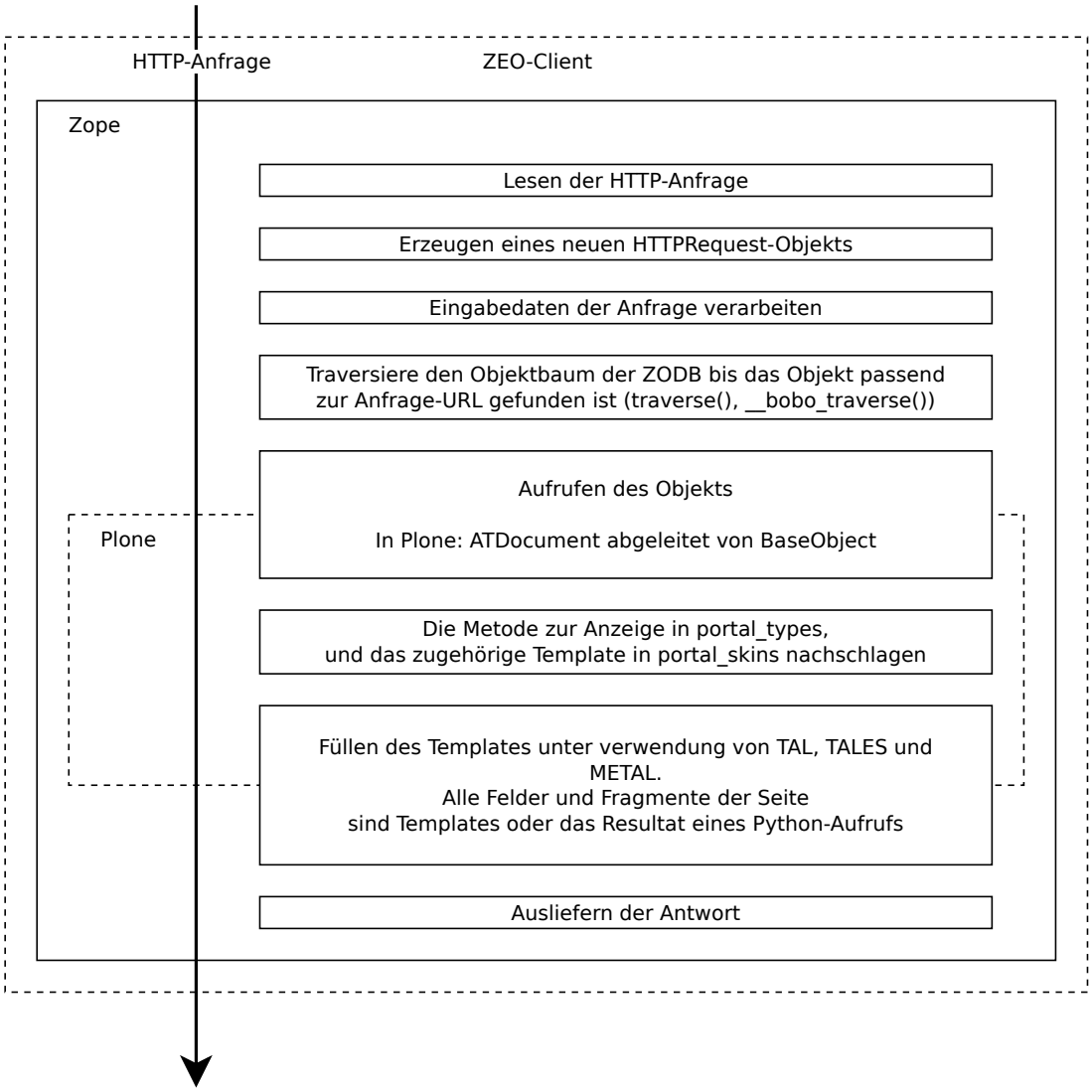

Abb. 4.2.: Stark vereinfachter Ablauf der HTTP-Anfragen innerhalb eines Zope-Webanwendungsservers mit installiertem Plone-Portal.

Daher muss für den Administrator auch die Möglichkeit bestehen, gegebenenfalls außerplanmäßig eine Aktualisierung des Index auslösen zu können. Dies kann dadurch geschehen, dass `getXML.py` auf der Kommandozeile aufgerufen wird. Es kann auch nur ein bestimmtes Plugin oder Portal aktualisiert werden, indem die Parameter `-p` und `-s` verwendet werden. Ersterer gibt den Namen des Plugins, Letzterer den Namen der Quelle an. Möchte man die Seiten unter `www.uni-freiburg.de` neu indexieren, so kann man dies mit `./getXML.py -p portal -s uni` tun. Die Daten der Forschungsdatenbank werden hingegen mit `./getXML.py -p forschdb -s forschdb` auf den neuesten Stand gebracht.

```
MAILTO="webmaster@uni-freiburg.de"
# m h      dom mon dow   command
  1 2,14   *   *   *     cd ~/dipl/completesearch/databases/unifr/
↪   buildxml/ && ./getXML.py
```

Abb. 4.3.: Crontab-Eintrag zum Starten des Skripts `getXML.py` um 8 und um 20 Uhr.

Backend: Das Backend kann mit dem Kommandozeilenparameter `-r` gestartet werden, der bewirkt, dass der Server neu gestartet wird, sollte er nicht normal beendet worden sein. Für den Fall, dass der Mechanismus versagt, oder der Rechner neu gestartet wurde, führt ein *Crontab*-Eintrag (siehe Abbildung 4.4) jede Minute ein Bash-Skript aus (siehe Abbildung 4.5), welches überprüft ob der Server noch läuft und ihn andernfalls neu startet.

```
MAILTO="webmaster@uni-freiburg.de"
# m h  dom mon dow   command
  * *  *   *   *     ~/bin/ensure_running_server.sh
```

Abb. 4.4.: Crontab-Eintrag, um zu überprüfen ob der CompletionServer noch läuft.

Der CompletionServer speichert seine Prozess-ID (*PID*) in einer Datei mit dem Schema `~/.completesearch_<Name des Servers>_<Port>.pid`. Diese wird vom Skript ausgelesen und mit `kill -0 $PID` getestet ob der Prozess noch läuft. Falls nicht, wird der CompletionServer neu gestartet und eine Meldung als Mail an die in `MAILTO` angegebene Adresse versandt (siehe Abbildung 4.6).

Anfrage-Webserver und Benutzerschnittstelle: Alle Fehler werden in die Logdatei geschrieben.[6] Sollte das Backend wider erwarten nicht erreichbar sein, wird diese Tatsache durch eine Fehlermeldung dem Benutzer mitgeteilt und auf *Google* als

[6] Siehe hierzu auch den folgenden Abschnitt 4.3.

```
#!/bin/bash
PID=‘cat ~/.completesearch_localhost_12345.pid‘
/bin/kill -0 $PID
if [[ $? -gt 0 ]]
then
  date >&2
  echo "Server not running! Starting..." >&2
  echo "Changing to directory '~/dipl/completesearch/databases/
  ↪   unifr'" >&2
  cd ~/dipl/completesearch/databases/unifr
  echo "Executing './codebase/server/startCompletionServer -r -p
  ↪   12345 -l unifr.log unifr.hybrid >&2'" >&2
  ./codebase/server/startCompletionServer -r -p 12345 -l unifr.log
  ↪   unifr.hybrid >&2
fi
```

Abb. 4.5.: Bash-Skript zum Überprüfen, ob der CompletionServer noch läuft. Wird von 4.4 jede Minute aufgerufen.

```
kill: No such process
Sat Sep  4 18:46:01 CEST 2010
Server not running! Starting...
Changing to directory '~/dipl/completesearch/databases/unifr'
Executing './codebase/server/startCompletionServer -r -p 12345 -l
↪   unifr.log unifr.hybrid >&2'
! getenv: variable HOSTNAME not found; using "localhost"
Forked server, all further output in "unifr.log"
```

Abb. 4.6.: Inhalt der Mail, welche versendet wird, wenn der CompletionServer durch das Skript aus Abbildung 4.5 neu gestartet wurde.

Alternative verlinkt. Dabei ist der Link mit den vom Benutzer ursprünglich eingegebenen Suchbegriffen und einer Einschränkung auf die Domain `uni-freiburg.de` vorbelegt (siehe Abbildung 4.7).

Sollte es Probleme mit dem Anzeigen oder der Verwendung der Suchseite geben, kann sich der Benutzer an den zuständigen Webmaster wenden. Ein Verweis auf die Kontaktmöglichkeiten finden sich am unteren Ende jeder Seite unter *Kontakt*.

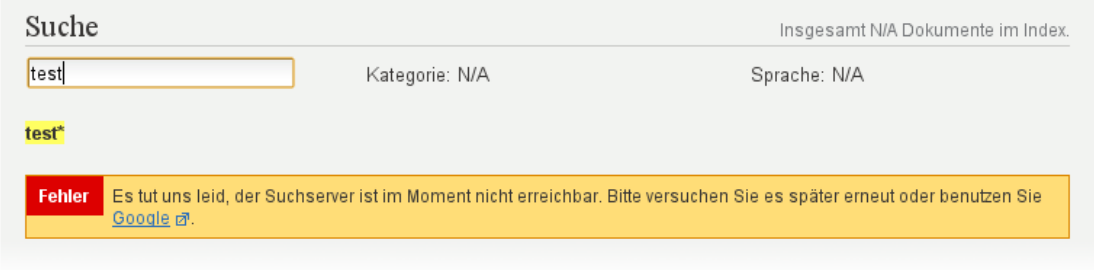

Abb. 4.7.: Fehlermeldung bei nicht erreichbarem Suchserver und Verlinkung zu Google mit den vom Benutzer eingegebenen Suchbegriffen, sowie Einschränkung auf die Domain `uni-freiburg.de`.

4.3. Logging und Debugging

Alle fünf Komponenten der neuen Suchfunktion speichern Meldungen in Logdateien, um Fehler und unerwünschtes Verhalten nachvollziehen, sowie gegebenenfalls statistische Auswertungen durchführen zu können.

Datenerfassung: Die Logdatei der Datenerfassung ist unter `./log/getXML.log` zu finden. Darin finden sich Einträge der verschiedenen Plugins, des Controllers und des Skripts `getXML.py`. Alle Einträge haben eines der *Logging-Level* `DEBUG`, `INFO`, `WARNING`, `ERROR` oder `CRITICAL`. Das Format ist wie folgt:

```
<Datum> <Zeit> - <Loglevel> - [<Klassenname>, <Name der Quelle>] -
↪   [<Datei>, <Zeile>] - <Nachricht>
```

Dabei ist `<Name der Quelle>` der in `buildxml.config` unter `PORTALS` oder `PLUGINS` eingetragene `name`. Über die Variable `LOG_LEVEL` in `buildxml.config` wird gesteuert, ab welchem Level die Meldungen in die Logdatei geschrieben werden. Dabei sind den Leveln in obiger Reihenfolge aufsteigende Werte zugeordnet. So werden bei Logging-Level `DEBUG` alle Meldungen geschrieben, während bei Logging-Level `WARNING` nur als Warnungen, Fehler oder kritischer Fehler eingestufte Meldungen geschrieben werden.

Für alle Logging-Level außer `DEBUG`, ist durch die Variable `LOG_ROLLOVER_SIZE` in `buildxml.config` die Größe der Logdatei in Bytes angegeben, ab welcher die

Logdatei *überläuft*[7] und eine neue Logdatei begonnen werden soll. Es werden bis zu `LOG_BACKUP_COUNT` alte Logdateien behalten, welche einen Dateinamen der Form `getXML.log.X` haben. Dabei ist `X` eine Nummer von 1 bis 9, wobei ältere Meldungen in Logdateien mit größerem `X` zu finden sind.

Alle Klassen, welche in die Logdatei schreiben sollen, müssen von der Klasse `BaseLogger` in `xmlgetter.log` abgeleitet werden, und verfügen dann über ein Attribut des Typs `logging.Logger`. Über dessen, zum jeweiligen Debug-Level korrespondierenden Funktionen, können dann Meldungen in die Logdatei geschrieben werden:

```
self.logger.debug(u'Meldung mit dem Level DEBUG')
self.logger.info(u'Meldung mit dem Level INFO')
self.logger.warning(u'Meldung mit dem Level WARNING')
self.logger.error(u'Meldung mit dem Level ERROR')
self.logger.critical(u'Meldung mit dem Level CRITICAL')
try:
    [...]
except:
    self.logger.exception(u'Meldung mit dem Level ERROR')
    # exception() sollte nur in einem except-Block verwendet
    # werden. Die Funktion gibt zusaetzlich zur Meldung einen
    # Traceback mit aus.
```

Die Datenerfassung versendet bei auftretenden Fehlern E-Mails. Daher ist es wichtig, in regelmäßigen Abständen das Postfach auf E-Mails mit Fehlermeldungen zu überprüfen, um die Aktualität und Vollständigkeit des Index zu gewährleisten. Sollte ein Fehler gemeldet werden, muss zuerst geprüft werden, ob der Fehler in der Logdatei nachzuvollziehen ist. Falls nicht, ist zu überprüfen, ob `LOG_LEVEL` auf `DEBUG` gesetzt ist und gegebenenfalls die Datenerfassung mit diesem Wert von Hand erneut zu starten. Falls weiterhin ungenügende Informationen in der Logdatei vorliegen, können dem Quelltext solange Anweisungen für zusätzliche Debugmeldungen hinzugefügt, und die Datenerfassung erneut gestartet werden, bis es möglich ist, den Fehler in der Logdatei einzugrenzen.

Für das Debugging der laufenden Datenerfassung empfiehlt sich der Python-Debugger `pdb`. Dazu kann an beliebiger Stelle im Quelltext ein

```
import pdb; pdb.set_trace()
```

eingefügt werden. Innerhalb der Debug-Konsole kann dann auf alle Variablen zugegriffen werden, die an der entsprechenden Stelle im Quelltext verfügbar sind – zum Beispiel, um sie mit `print <variable>` auszugeben. Näheres zu `pdb` und seiner Verwendung findet sich in [6].

[7] Aus dem Englischen: *rollover*.

Die Klassen `XMLEntry`, `Stats` und `PortalSourceState` überladen die Funktion `__str__()` so, dass Instanzen dieser Klassen direkt geloggt oder per `print` ausgegeben werden können. Objekte der Klasse `Stats` werden dazu verwendet, eine tabellarische Übersicht über die statistischen Informationen eines Plugins zu erstellen.

```
[Source]   [Static] [Entries] [New]   [Modified] [Unmod.] [Status]
  forschdb   83734    83734     0       0          0        W
  stb        0        0         0       0          0        F
  studium    184      3696      0       0          3512     ok

There are messages:

[forschdb]
  WARNING: 2 consecutive unsuccessful attempts to reach the server!
[stb]
  FAILED: Could not reach the server!

Times [h:m:s.ms]:
Total: 0:14:01.021009
[forschdb] : 0:11:26.112837
[stb] : 0:00:00.795335
[studium] : 0:02:34.112837
```

Abb. 4.8.: Statistik über die Abgefragten Quellen während der Datenerfassung. Die Spalte `Static` gibt an, wie viele Einträge der Quelle das Aktualisieren nach Datum nicht unterstützen und daher als *statisch* angesehen werden, d.h. nicht als neu oder modifiziert gezählt werden.

Wie in Abbildung 4.8 zu sehen ist, können auch Meldungen für jedes Plugin ausgegeben werden. Die Meldungen können über die Funktion `addMessage()` der Klasse `Stats` hinzugefügt werden. In der Textrepräsentation der Klasse werden die Meldungen dann unterhalb der Statistik mit ausgegeben.

Indexierung und Backend: Während den Phasen der *Indexierung* (Parsen, Sortieren und Erzeugen des binären Suchindex) werden unter anderem Informationen über das Volumen, Anzahl der Wörter, Blockgrößen im Index und die Größe der resultierenden *HYB*-Datenstruktur ausgegeben. Weiterhin werden während der Indexerzeugung aufgetretene Fehler in die Datei `.unifr.hybrid.build-index-errors` im Verzeichnis des Parsers, sowie weitere Meldungen nach `.unifr.hybrid.build-index-log` ausgegeben.[8]

Alle Suchanfragen, Fehler und Meldungen werden vom **Backend** in der Datei gespeichert, die über den Parameter `-l` übergeben wurde. In diesem Fall `unifr.log`

[8] Siehe in Abbildung 3.9 die Verzeichnisstruktur und relevante Dateien.

im Verzeichnis `databases/unifr`[9]. Die Verbosität des CompletionServers kann erhöht werden, indem er mit dem Parameter `--verbosity=2` gestartet wird.

Mit einem geeigneten Debugger können sowohl Fehler in der Indexierung als auch im Backend gefunden werden. Hierzu bietet sich beispielsweise der *GNU-Projekt-Debugger* (*GDB*)[10] an. Auch empfiehlt sich oftmals ein separates Starten des Parsens und Indexierens durch eine sequenzielle manuelle Eingabe von `make parse`, `make sort` und `make index` in der Konsole. Die Befehle des Makefile können jeweils mit dem Parameter `-n` ausgegeben werden, ohne dass diese tatsächlich ausgeführt werden. Dies erleichtert die Fehleranalyse und erlaubt das Ausführen einzelner Teilbefehle zur weiteren Fehlereingrenzung.

Anfrage-Webserver: Alle Anfragen an den CompletionServer werden über die Klassen `CSQuery` und `CSRequest` durchgeführt. Jede dieser Anfragen werden in der *Logdatei* unter `var/log/complete.log`, ausgehend vom Startverzeichnis des ZEO-Clients protokolliert.[11] Durch die Anfrage-ID[12] können diese in der Logdatei identifiziert werden. Als ID wird die Systemzeit in Millisekunden verwendet, wie sie durch die Funktion `time()` des Moduls `time` zurückgegeben wird. Zusätzlich wird die anonymisierte IP-Adresse des anfragenden Benutzers protokolliert, um die Anfrage zurückverfolgen und statistische Informationen sammeln zu können.

Zu Zwecken der Optimierung[13] und Fehlerverfolgung werden für jede Anfrage Zeitmessungen durchgeführt. Es werden die Zeit, welche der CompletionServer für die Beantwortung der Anfrage benötigt (`A`), die akkumulierten Zeiten für die Kommunikation zwischen Anfrage-Webserver und Server (`B`) sowie die Skriptlaufzeit von `complete` (`C`) gemessen und in einem Objekt der Klasse `CSTimingInfo` gespeichert. Alle Werte der Zeitmessungen werden an die Benutzerschnittstelle weitergeleitet und dort um die Werte der Zeitmessungen für die Kommunikation zwischen Anfrage-Webserver und Benutzerschnittstelle (`D`) und die Laufzeit der JavaScript-Funktionen (`E`) ergänzt. Nach erfolgreichem Anzeigen der Suchergebnisse werden die Messergebnisse über die Funktion `logFinalMessage()` in einer zusammenfassenden Zeile in die Datei `complete.log` geschrieben. Ein Beispiel einer solchen zusammenfassenden Zeile mit allen Werten ist in Abbildung 4.9 gegeben. Alle Zeiten werden in Millisekunden angegeben. Der Wert `F` steht für die akkumulierte Zeit vom Absenden der Anfrage bis zum Anzeigen der Ergebnisse. Für das gegebene Beispiel ist zu beachten, dass der Anfrage-Webserver auf einem Laptop außerhalb des Netzwerks der Universität

[9] Siehe Abschnitt 3.4.1 und Abbildung 3.9.
[10] *GDB: The GNU Project Debugger*, `http://www.gnu.org/software/gdb/`.
[11] Siehe auch Abschnitt 4.3.
[12] Attribut `_query_id` der Klasse `CSQuery`.
[13] Ausführlicher dargestellt in Abschnitt 3.7.

```
2010-05-23 19:29:46,901 - complete.py - line 223 - INFO
[STATS] for query 1274635786357.3420410:
  127.0.0.XXX | q=test*&h=10&f=0&er=10&en=1&c=20 | A=31.7 | B=351 |
↪    C=82.9 | D=133.4 | E=26.0 | F=625.0
```

Abb. 4.9.: Zusammenfassender Eintrag der Zeitmessungen des Anfrage-Webservers und der Benutzerschnittstelle in der Datei `complete.log`. Die Zeiten für die Kommunikation der beteiligten Komponenten ist stark erhöht, da die Messung nicht in der Produktivumgebung durchgeführt wurde.

lief und daher die Zeiten für die Kommunikation und die Zeiten für die Verarbeitung durch `complete` teilweise erheblich höher sind, als in der Produktivumgebung.

Die Zope-Erweiterung `complete.py`, welche die externen Methoden für den Anfrage-Webserver bereitstellt, loggt über die globale Variable mit dem Namen `_logger` des Typs `logging.Logger`. Diese Variable wird als Attribut einer Instanz von `CSLogger` bereitgestellt. Auch hier wird, wie schon bei der Datenerfassung, ein `RotatingFileHandler` verwendet, um die Dateigröße zu begrenzen und bis zu neun Backups vorzuhalten. Die Klasse `CSLogger` implementiert ein Singleton-Pattern, um eine mehrfache Instanziierung und damit doppelte Logmeldungen zu vermeiden.

Wie auch bei der Datenerfassung kann das Logging-Level eingestellt werden. Dazu ist die Variable `LOG_LEVEL` im Skript `complete.py` entsprechend anzupassen. Dort können auch ein anderer Pfad und Name für die Logdatei angegeben werden. Außerdem ist hier die URL des Backends angegeben, und es kann der Timeout für die HTTP-Anfragen[14] eingestellt werden.

Der Debug-Modus des ZEO-Clients kann mit `./zeo1 debug`[15] gestartet werden. Innerhalb des Debug-Modus steht eine interaktive Python-Konsole zur Verfügung, die vollen Zugriff auf die ZODB erlaubt – inklusive des Plone-Portals und aller in ihm existierenden Objekte.

Weiterhin können auch in `complete.py` Breakpoints, wie schon oben für die Datenerfassung beschrieben, gesetzt werden. Nach dem Starten des ZEO-Clients im Vordergrund mit `./zeo1 fg` wird eine interaktive `pdb`-Konsole gestartet, sobald der Breakpoint erreicht ist. Hilfreich ist hier auch, dass die zentralen Klassen `CSQuery`, `CSRequest`, `CSHit`, `CSCompletion` und `CSResponse` eine überschriebene `__str__()`-Funktion besitzen, die eine Ausgabe über die `print`-Funktion erlaubt. Für ein Beispiel einer `pdb`-Sitzung, bei der eine Variable des Typs `CSQuery` ausgegeben wird, ist in Abbildung 4.10 zu sehen.

[14] Vorgabewert ist 60 Sekunden.
[15] Der Dateiname des Startskripts des ZEO-Clients kann abweichend auch `client1` o.ä. lauten.

```
> /home/johannes/Plone/Zope-2.10.11-final-py2.4/Extensions/complete
↪   .py(2632)query()
-> timing = CSTimingInfo()
(Pdb)
(Pdb) dir()
['additional_filters', 'category_filter', 'context', 'first_call',
↪   'html', 'ip', 'language_filter', 'limit', 'pdb', 'q', 'query',
↪    'request', 'show_from']
(Pdb) print query
[QUERY] 1283534792174.6530762 from 127.0.0.XXX : q=test*&h=10&f=0&
↪   er=10&en=1&c=20
        Messages:
        000.9ms, 000.9ms - Parameters parsed
        000.0ms, 000.9ms - Query parameters assigned
(Pdb)
```

Abb. 4.10.: Beispiel einer `pdb`-Sitzung für das Skript `complete.py`.

Benutzerschnittstelle: Um das Verfolgen der Suchanfragen und deren Zeitmessungen zu vereinfachen, wird für einen angemeldeten Benutzer eine Konsole eingeblendet, welche die Meldungen für die Suchanfrage[16], die zusammenfassende Zeile mit allen wichtigen Zeitmessungen und Meldungen des JavaScript `complete.js` anzeigt (siehe Abbildung 4.11).

Die Konsole wird durch das Einbinden des JavaScripts `dragresize.js` bereitgestellt, und kann mit der Maus sowohl in ihrer Größe, als auch in der Position beliebig angepasst werden. Das Meldungsfenster zeigt eine Bildlaufleiste an, sobald die Nachrichten die Höhe des Anzeigebereichs überschreiten und scrollt automatisch nach unten. Das verwendete JavaScript und die HTML-Struktur der Suchseite können außerdem durch Einsatz einer JavaScript-Konsole und Entwicklerwerkzeugen – wie etwa die des Browsers *Chromium* oder *FireBug* für *Firefox* – auf Fehler untersucht werden.

4.4. Quelltext und Dokumentation

Alle Quelltexte der Datenerfassung, des Anfrage-Webservers und deren HTML-Dokumentation sowie die Quelltexte der Benutzerschnittstelle sind im *SVN*-Repository unter `http://vulcano.informatik.uni-freiburg.de/svn/completesearch-uni-freiburg` zu finden.

[16] Meldungen die dem `query`-Objekt in den Funktionen in `complete.py` durch den Aufruf von `addMessage()` hinzugefügt wurden.

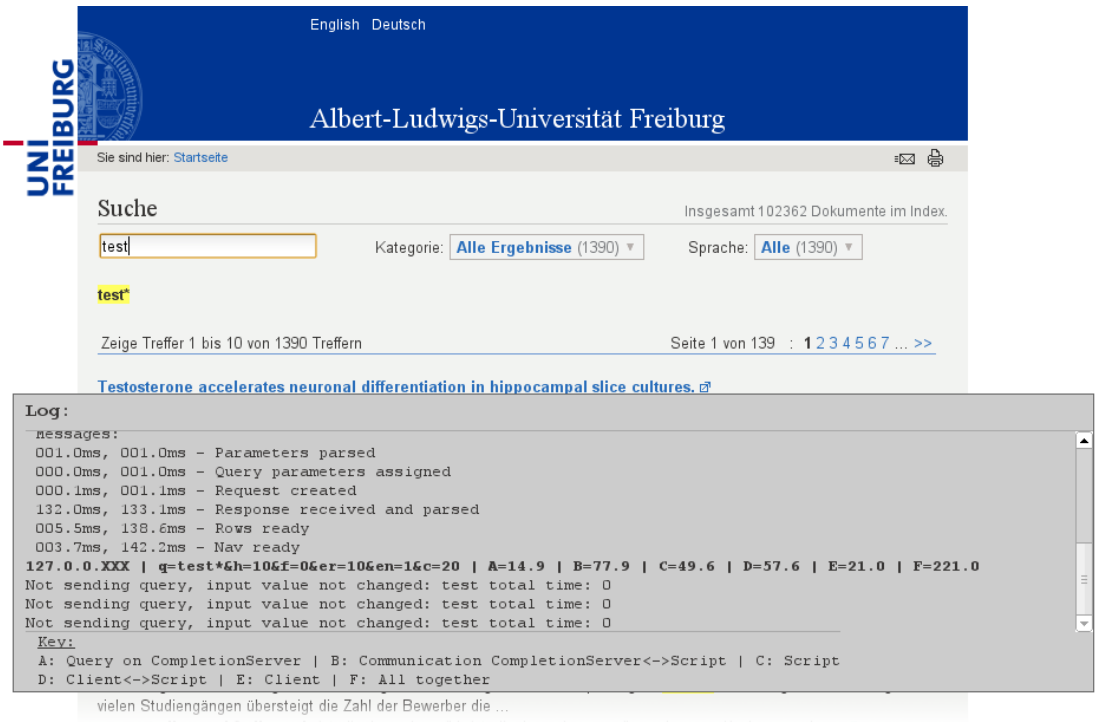

Abb. 4.11.: Konsole für das Anzeigen der Suchanfragen und der Zeitmessungen.

Die PDF-Version dieser Diplomarbeit ist unter `http://stromboli.informatik.uni-freiburg.de/student-projects/johannes-schwenk/` zu finden. Außerdem kann dort die Dokumentation der Quelltexte unter Verwendung des Internetbrowsers gelesen werden.

5. Fazit

Die im Rahmen dieser Arbeit entwickelten Komponenten zur *Datenerfassung*, *Indexierung*, der *Anfrage-Webserver* und die *Benutzerschnittstelle* ergeben im Zusammenspiel mit dem *Backend* eine benutzerfreundliche Suchfunktion, mit der das quellenübergreifende Suchen über mehrere Subdomains der Universität möglich ist. Dabei wurden repräsentative Quellen mit insgesamt über 90.000 Datensätzen ausgewählt und ein Plugin-System zum einfachen Erweitern der Datenbasis entwickelt. Die entstandene Sofortsuche kann über ein einziges Suchfeld bedient werden und erlaubt das Verfeinern der Ergebnisse über jeweils ein Menü für die Kategorie und und eines für die Zielsprache der durchsuchten Inhalte.

Für Inhalte vergebene Kategorien (Tags) innerhalb Plone werden als Wort mit der hundertfachen Gewichtung für den jeweiligen Eintrag im Index eingefügt. Dadurch kann die Rangfolge der Suchergebnisse beeinflusst werden.

Es wurden laufend Optimierungen durchgeführt, um die Laufzeit sowohl für die *Datenerfassung*, als auch für den *Anfrage-Webserver* zu minimieren. Die *Datenerfassung* wird zweimal täglich automatisch gestartet und aktualisiert innerhalb von etwa 25 Minuten den Index. Suchergebnisse werden innerhalb von durchschnittlich einer Drittelsekunde angezeigt. Die Zeiten für den Aktualisierungsprozess und die einzelnen Suchanfragen, sowie die Suchanfragen selbst, nebst den anonymisierten IP-Adressen der Benutzer, werden in Logdateien für statistische Auswertungen und das Auffinden von eventuellen Fehlern festgehalten.

Tritt ein Fehler bei der *Datenerfassung* auf, wird eine Benachrichtigung per E-Mail an den zuständigen Administrator versandt. Sollte das *Backend* nicht erreichbar sein, wird der Benutzer über eine Meldung auf der Suchseite darauf hingewiesen und über einen Link auf Google als externen Suchanbieter verwiesen. Ein Cronjob überwacht laufend den CompletionServer und startet diesen bei Bedarf neu – auch in diesem Fall wird der Administrator per E-Mail benachrichtigt.

5.1. Ausblick

Mit den bisher indexierten Datenquellen kann die neue Suchfunktion den Einsatz eines externen Suchanbieters wie Google noch nicht vollständig überflüssig machen. Um dies zu erreichen, müssen weitere Quellen hinzugefügt werden, was aufgrund

des implementierten Plugin-Systems keinen allzu großen Aufwand darstellen sollte, aber leider im zeitlichen Rahmen dieser Diplomarbeit nicht umzusetzen war. Einige wichtige Quellen sind beispielsweise die Lehrveranstaltungsmanagement-Software LSF (Lehre Studium Forschung) mit dem Vorlesungsverzeichnis, die Kataloge der Universitätsbibliothek, die Seiten der Zentralen Universitätsverwaltung, die Seiten des Rechenzentrums und die Seiten des Freiburg Institute for Advanced Studies (FRIAS).

Die Basisklasse des XML-Parsers für die Indexierung sollte noch dahingehend angepasst werden, dass auch die *Synonym-Suche* verfügbar ist. Alternativ kann das System auch auf den Parser für Comma Seperated Values (CSV) umgestellt werden, welcher die Synonym-Suche bereits unterstützt. Da keine geschachtelten XML-Tags verwendet werden, könnte während der Datenerfassung eine neue Klasse `CSVEntry` dafür sorgen, dass statt den XML-Daten kommaseparierte Daten geschrieben werden. Überdies könnte dann der Schritt des Validierens der XML-Datei mit `xmllint` am Ende der *Datenerfassung* entfallen.

Der reibungslose Betrieb und die Verfügbarkeit der Suchfunktion ist von großer Wichtigkeit, wenn die neue Suchfunktion als einzige Möglichkeit zum Auffinden von Inhalten angeboten werden soll. Leider reichte auch hier die vorgegebene Zeit nicht aus, um intensive Tests über einen längeren Zeitraum durchzuführen. Daher sollte eine längere Testphase dem produktiven Einsatz vorausgehen, in welcher eventuelle Fehler identifiziert und behoben werden können.

Die bisherige Suche stellt ein Portlet für die schnelle Einbindung auf jeder im Plone-Portal vorhandenen Seite bereit. Obwohl ein solches Portlet bisher nicht implementiert ist, dürfte es keine Schwierigkeit sein das bestehende Portlet umzuarbeiten, sodass es statt mit dem lokalen `portal_catalog` über die entwickelten Erweiterungen des Webservers mit dem *Backend* kommuniziert.

Ist die neue Suchfunktion in ihrer Funktionalität komplett und sind alle Tests zur Zufriedenheit abgeschlossen, sollte die Möglichkeit in Erwägung gezogen werden, alle zugehörigen Skripte, Templates und die Erweiterungen des Webservers zur Bearbeitung der Anfragen an das *Backend* in einem Plone-*Produkt* zu vereinigen. Dieses Produkt kann dann über den Plone-eigenen Mechanismus installiert und aktualisiert werden, wodurch die Administration weiter vereinfacht wird. Das Skript `remoteSyncQueryXML` könnte ebenfalls in einem Produkt bereitgestellt werden, wodurch auch hier die Aktualisierung auf neuere Versionen vereinfacht wird.

Sollte bei wachsender Anzahl der indexierten Quellen die Laufzeit der *Datenerfassung* so stark steigen, dass eine zweimal tägliche Aktualisierung nicht mehr möglich ist, sollte man in Erwägung ziehen, jedes Plugin in einem eigenen *Thread* zu starten, um so durch *Parallelisierung* die Laufzeit zu verkürzen. Dabei sollte beachtet werden, dass Portale die über den selben ZEO-Client aktualisiert werden, nicht gleichzeitig

aufgerufen werden, um den Client nicht zu überlasten. Sollte diese Maßnahme zur *Optimierung* dennoch nicht ausreichen, kann der Nutzen einer Neu-Implementierung der *Datenerfassung* in einer hardwarenäheren Sprache gegen den Aufwand für die Portierung abgewogen werden.

Anhang

A.1. Klassendiagramme

A.1.1. Datenerfassung - buildxml

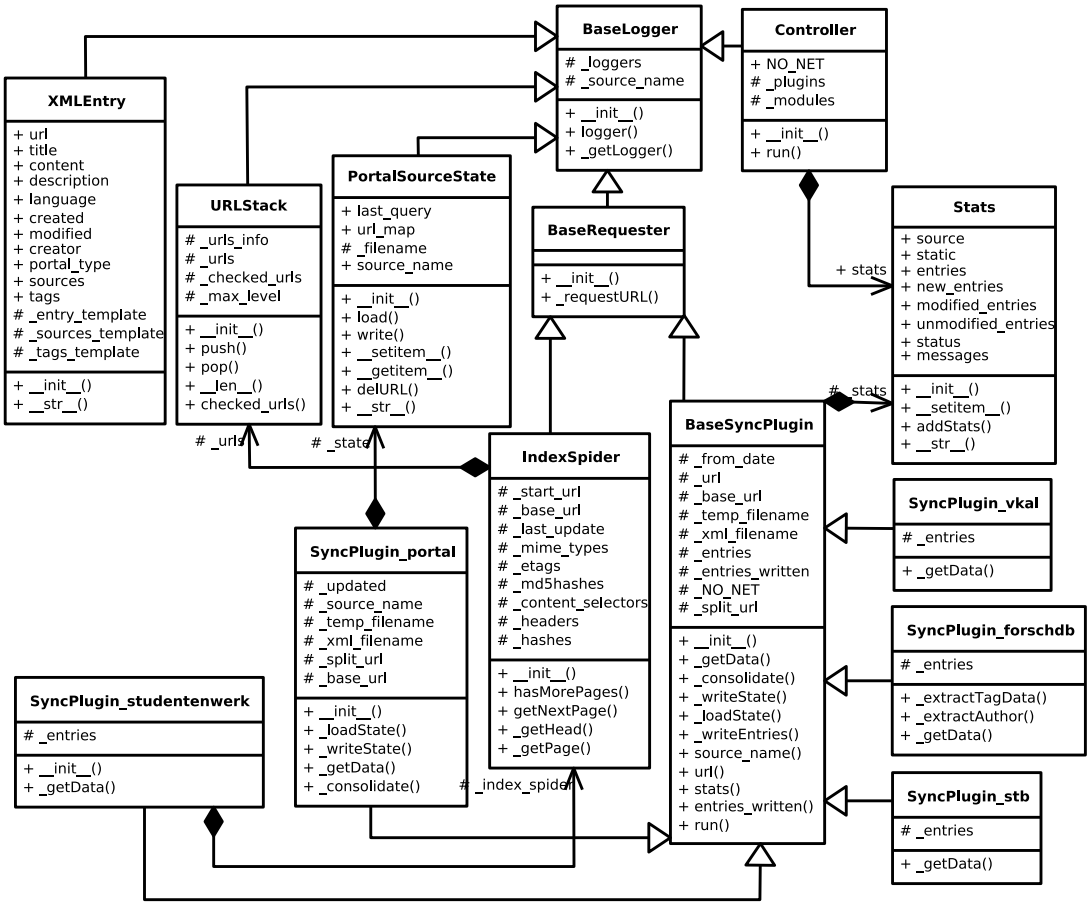

Abb. A.1.: Klassendiagramm `buildxml` ohne die Klasse `BeautifulSoup`

Für die Beschreibung der Datenerfassung siehe Abschnitt 3.2.

A.1.2. Anfrage-Webserver - complete

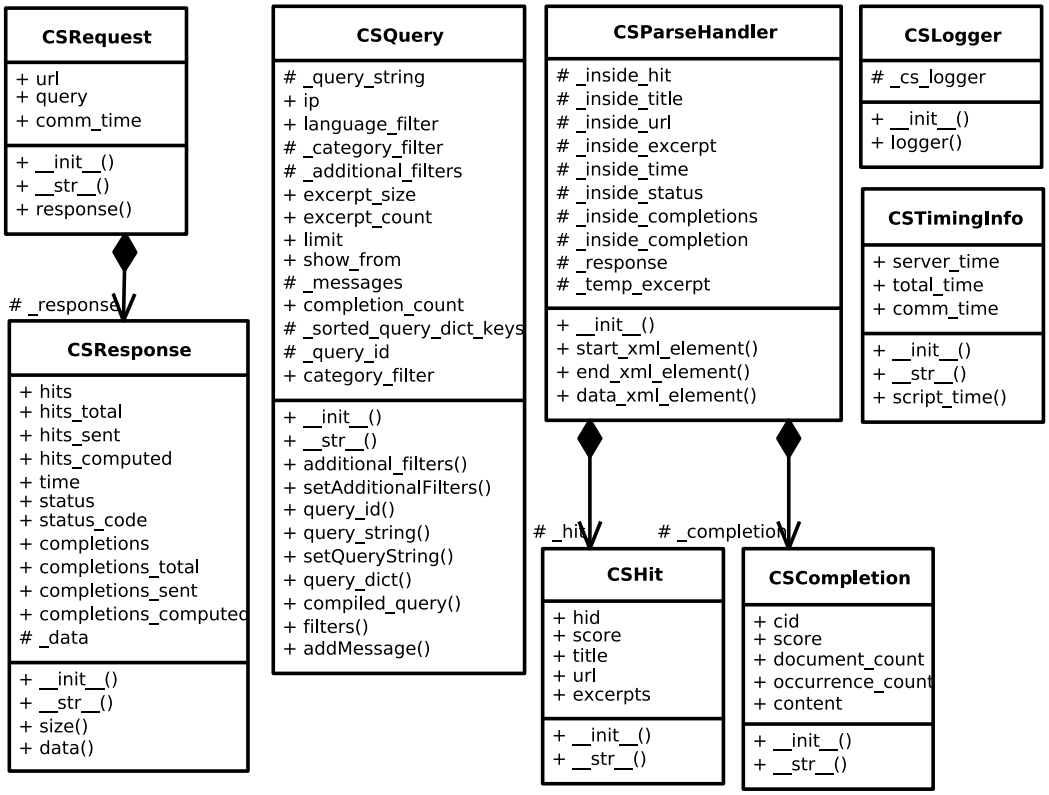

Abb. A.2.: Klassendiagramm `complete`

Für die Beschreibung des Zope-/Plone-Clients siehe Abschnitt 3.5.

Literaturverzeichnis

[1] *Appendix C: Zope Page Templates Reference*. http://www.zope.org/. `http://www.zope.org/Documentation/Books/ZopeBook/2_6Edition/AppendixC.stx`. Version: Jul. 2010. – Zugriff: 2010-07-02. (Archiviert von WebCite® unter http://www.webcitation.org/5quvV3ZkO)

[2] *Basic DTML*. http://www.zope.org/. `http://www.zope.org/Documentation/Books/ZopeBook/2_6Edition/DTML.stx`. Version: Jul. 2010. – Zugriff: 2010-07-27. (Archiviert von WebCite® unter http://www.webcitation.org/5rX8qXKRA)

[3] *Developing with Zope 2 - Acquisition*. http://wiki.zope.org/. `http://wiki.zope.org/zope2/Acquisition`. Version: Jul. 2010. – Zugriff: 2010-07-02. (Archiviert von WebCite® unter http://www.webcitation.org/5qvMxOc6z)

[4] *GPL-Compatible Free Software Licenses*. http://www.gnu.org/. `http://www.gnu.org/licenses/license-list.html#GPLCompatibleLicenses`. Version: Jun. 2010. – Zugriff: 2010-06-24. (Archiviert von WebCite® unter http://www.webcitation.org/5qiyI1vVj)

[5] *Homepage BeautifulSoup*. http://www.crummy.com/. `http://www.crummy.com/software/BeautifulSoup/`. Version: Jul. 2010. – Zugriff: 2010-07-07. (Archiviert von WebCite® unter http://www.webcitation.org/5r31FgNKc)

[6] *The Python Debugger*. http://docs.python.org/. `http://docs.python.org/library/pdb.html`. Version: Aug. 2010. – Zugriff: 2010-08-01. (Archiviert von WebCite® unter http://www.webcitation.org/5rdyeuRoS)

[7] *Python Performace Tips*. http://wiki.python.org/. `http://wiki.python.org/moin/PythonSpeed/PerformanceTips`. Version: Jul. 2010. – Zugriff: 2010-07-12. (Archiviert von WebCite® unter http://www.webcitation.org/5rAIxWQ1p)

[8] *Scalability and ZEO*. http://docs.zope.org/. `http://docs.zope.org/zope2/zope2book/ZEO.html`. Version: Jun. 2010. – Zugriff: 2010-06-25. (Archiviert von WebCite® unter http://www.webcitation.org/5qkOX4MCe)

[9] *Using External Methods*. http://docs.zope.org/. `http://docs.zope.org/zope2/zope2book/ScriptingZope.html#using-external-methods`. Version: Jun. 2010. – Zugriff: 2010-06-14. (Archiviert von WebCite® unter http://www.webcitation.org/5qU5F9izC)

[10] *Z SQL Methods User's Guide*. http://www.zope.org/. `http://www.zope.org/Documentation/Guides/ZSQL-HTML/ZSQL.html`. Version: Jul. 2010. – Zugriff: 2010-07-27. (Archiviert von WebCite® unter http://www.webcitation.org/5rX8R0JUn)

[11] *Zope Concepts and Architecture*. http://docs.zope.org/. `http://docs.zope.org/zope2/zope2book/ZopeArchitecture.html#fundamental-zope-components`. Version: Jun. 2010. – Zugriff: 2010-06-24. (Archiviert von WebCite® unter http://www.webcitation.org/5qizaHKR9)

[12] Anh, V. N. ; Moffat, A. : Inverted Index Compression Using Word-Aligned Binary Codes. In: *Inf. Retr.* 8 (2005), Nr. 1, S. 151–166

[13] Bast, H. ; Chitea, A. ; Suchanek, F. M. ; Weber, I. : ESTER: efficient search on text, entities, and relations. In: Kraaij, W. (Hrsg.) ; Vries, A. P. (Hrsg.) ; Clarke, C. L. A. (Hrsg.) ; Fuhr, N. (Hrsg.) ; Kando, N. (Hrsg.): *SIGIR*, ACM, 2007. – ISBN 978–1–59593–597–7, S. 671–678

[14] Bast, H. ; Weber, I. : Type less, find more: fast autocompletion search with a succinct index. In: Efthimiadis, E. N. (Hrsg.) ; Dumais, S. T. (Hrsg.) ; Hawking, D. (Hrsg.) ; Järvelin, K. (Hrsg.): *SIGIR*, ACM, 2006. – ISBN 1–59593–369–7, S. 364–371

[15] Bast, H. ; Weber, I. : When You're Lost For Words: Faceted Search With Autocompletion. In: Broder, A. (Hrsg.) ; Maarek, Y. (Hrsg.): *SIGIR'06 Workshop on Faceted Search*. Seattle, USA : ACM, August 2006, S. 31–35

[16] Bast, H. ; Weber, I. : The CompleteSearch Engine: Interactive, Efficient, and Towards IR& DB Integration. In: *CIDR*, www.crdrdb.org, 2007, S. 88–95

[17] Celikik, M. ; Bast, H. : Fast error-tolerant search on very large texts. In: Shin, S. Y. (Hrsg.) ; Ossowski, S. (Hrsg.): *SAC*, ACM, 2009. – ISBN 978–1–60558–166–8, S. 1724–1731

Tabellenverzeichnis

Abbildungsverzeichnis

Liste der Algorithmen