Wie kann in Millionen sehr kurzer OCR-Texte schnell und fehlertolerant gesucht werden?

Zur Indexierung eines großen Bibliothekskataloges

Eberhard Pietzsch*†

Zusammenfassung

In diesem Aufsatz wird ein Verfahren und seine Implementierung vorgestellt, wie in großen Mengen sehr kurzer OCR-Texte schnell und fehlertolerant recherchiert werden kann. Solche OCR-Texte entstehen beispielsweise bei der Digitalisierung älterer Bibliothekskataloge, die als Imagekataloge über Internet zugänglich gemacht werden, und bei denen die Images einer OCR-Texterkennung unterzogen werden. Das Verfahren soll dazu dienen, das Potential solcher Imagekataloge unter Verzicht auf manuelle Eingriffe voll auszuschöpfen. Einfache Implementierung und kurze Antwortzeiten sind wichtige Entwurfsziele.

1 Einleitung

Viele Datenbanken haben ihren Weg in das Intra- oder Internet gefunden. So mancher in früherer Zeit angelegte Papierkatalog ist jedoch via Internet unzugänglich, weil die Überführung in eine Datenbank nur mit erheblichem personellen Aufwand zu leisten wäre. Man denke an ältere, meist abgeschlossene, durchaus mehrere Millionen Einträge umfassende Zettelkataloge von Bibliotheken, Museen oder Wörterbuchverlagen. Die Konversion und Integration solcher Kataloge in zeitgemäße, für neuere Datenbestände oftmals schon vorhandene Datenbanken kann je nach Umfang und Komplexität durchaus mehrere Dutzend Personenjahre erfordern [5].

Während derartige Konversionsvorhaben hohe Kosten verursachen, ist Nichtstun ebenfalls nicht gratis zu haben: Der hergebrachte Papierkatalog kann nur aufwendig in orts- und zeitgebundener Form genutzt werden, falls er nicht völlig übersehen wird, weil fälschlicher Weise angenommen wird, die Recherche in einer via Internet zugänglichen Datenbank reiche aus. Wird der Katalog übersehen, so bleiben mit ihm auch die nachgewiesenen Bestände unentdeckt.

*Universitätsbibliothek Heidelberg, Email: pietzsch@uni-hd.de

†Der Autor dankt Gabriele Dörflinger und Leonhard Maylein für wertvolle Anregungen

Mancherorts werden daher Interimslösungen geschaffen, indem Papierkataloge eingescannt und die Images mit Ordnungshilfen im Internet bereitgestellt werden. Diese vergleichsweise preiswerten Imagekataloge bringen einen hohen Produktivitätsgewinn mit sich. Das Potential mancher Imagekataloge kann jedoch erst mit einer fehlertoleranten Recherche in den mittels OCR-Verfahren rekonstruierten Texten voll ausgeschöpft werden. Als reine Softwarelösung ist diese zudem preiswert.

Geht man von dem Gedanken aus, daß der Einsatz von Imagekatalogen als Interimslösung bis zur datenbankbasierten Neuerfassung des jeweils zu Grunde liegenden Kataloges zeitlich befristet ist, unterliegen Imagekataloge besonderen Wirtschaftlichkeitsanforderungen: Unter Vermeidung teurer manueller Arbeiten soll eine effektive Nutzung erreicht werden, die sich beispielsweise in kurzen Aufenthaltsdauern der Benutzer widerspiegelt.

In diesem Aufsatz wird beschrieben, wie die große Menge sehr kurzer Texte, die bei der OCR-Behandlung solcher Kataloge entsteht, indexiert werden kann, um fehlertolerant darin recherchieren zu können. Dabei wird dem Laufzeitverhalten bei der Recherche besondere Aufmerksamtkeit gewidmet. Auf andere Retrievalmethoden, die allesamt eine manuelle und daher kostenintensive Teil- oder Vollindexierung der Ordnungsbegriffe voraussetzen, wird nicht eingegangen, weil diese bereits an anderer Stelle (z.B. [1, 5]) hinreichend beschrieben sind.

Zur Implementierung des Verfahrens wird die im Web verbreitete Sprache Perl[1] sowie das relationale Datenbankmanagementsystem MySQL[2] verwendet. Die Software ist daher plattformunabhängig. Zur Erzielung kurzer Antwortzeiten wird zum einem die Perl Data Language[3] eingesetzt und zum anderen die Parallelität von Bearbeitungsprozessen und Benutzerinteraktion genutzt. Dadurch werden im Ergebnis kaum spürbare Antwortzeiten erreicht.

2 Das Material

Wir betrachten multilinguale Papierkataloge im Umfang von bis zu mehreren Millionen Karten K_i (Abb. 1). Ist n die Zahl der Dokumente (Karten), so seien die Images I_i der Dokumente K_i fortlaufend durchnummeriert: $I_i \in \{I_0, \ldots, I_{n-1}\}$. Ein I_i repräsentiert ein Dokument K_i. Die I_i werden einer OCR-Texterkennung unterzogen. Dabei entstehen Textdateien (*OCR-Texte*). Die Wörter in einem OCR-Text entsprechen wegen der OCR-Erkennungsfehler oft nicht denen im repräsentierten Dokument. Der Grad dieser Nicht-Übereinstimmung hängt wesentlich von der visuellen Qualität der Vorlagen ab. Diese kann bei historisch gewachsenen und über Jahre benutzten Katalogen durch unterschiedliche Phänomene stark beeinträchtigt sein:

[1] `http://www.perl.org`
[2] `http://www.mysql.com`
[3] `http://pdl.perl.org`

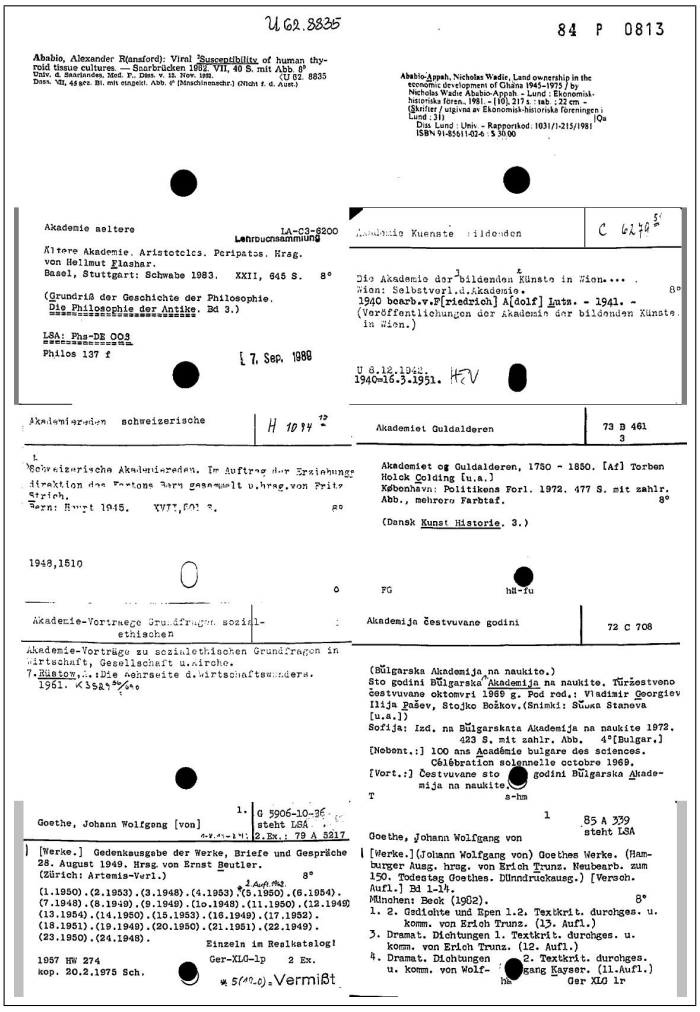

Abbildung 1: Eine kleine Auswahl von Karten eines älteren Bibliothekskataloges läßt erahnen, wie unterschiedlich die Abbildungsqualität sein kann.

Schriftarten. Verschiedene Schreibwerkzeuge können Ursache für unterschiedliche Schriftgrößen und -arten sein.

Ergänzungen. Manche Karten können maschinen- oder handschriftlich ergänzt worden sein, und zwar mit wiederum abweichenden Schriftarten.

Kontraste. Die Images können untereinander, aber auch in sich unterschiedliche Schwärzungen und Kontraste aufweisen.

Abnutzung. Verschmutzungen, unterschiedlich starke Beanspruchung oder mechanische Verletzungen können die Lesbarkeit beeinträchtigen.

Sprachen. In multilingualen Katalogen werden mehrere Sprachen verwendet, z.T. auch innerhalb einzelner Karten. In Katalogen großer Bibliotheken können durchaus mehrere Dutzend west- und osteuropäischer Sprachen verwendet sein.

All dies trägt zu einer heterogenen Texterkennungsqualität bei. OCR-Verfahren können daher hohe Wortfehlerraten aufweisen. Schäuble [4] ermittelte bei einem großen Bibliothekskatalog eine durchschnittliche Wortfehlerrate von etwa 33%, die durch eigene Tests [5] verifiziert werden konnte.

Die Erkennungsqualität kann insgesamt verbessert werden, wenn mehrere Texterkennungsverfahren unabhängig voneinander angewendet werden. Je nach Vorlage reichen die Ergebnisse jedoch auch dann nicht an die Erkennungsqualität bei gleichförmigen Vorlagen heran.

Fehlerkorrekturverfahren von OCR-Programmen gleichen den erkannten Text mit Wörterbüchern ab und können so die Zahl der Wortfehler verringern. Derartige Verfahren sind bei den betrachteten multilingualen Vorlagen nur von begrenztem Nutzen. Darüber hinaus sind sie bei Eigennamen wirkungslos.

Die entstehenden OCR-Texte sind sehr kurz. Eine Karte enthält durchschnittlich etwa zwei Dutzend Wörter. Das gewünschte Retrievalverfahren zielt darauf ab, zu gesuchten Stichworten (*Query*) $\Phi = \{\varphi_0, \ldots, \varphi_{m-1}\}$ die Nummern der Dokumente K_i zu finden, die diese Stichworte sämtlich oder teilweise enthalten. Dazu werden die OCR-Texte indexiert und ein Volltextretrieval soll unter Berücksichtigung der Wortfehler das gewünschte Ergebnis als Rangliste liefern. Es ist nicht erforderlich, jedes Vorkommen eines Wortes in einem einzelnen Text zu lokalisieren.

Längere OCR-Texte weisen Redundanz auf. Relevante Wörter erscheinen in der Regel vielfach. Hier sind auch Retrievalverfahren ohne Fehlertoleranz erfolgversprechend, weil nur eines der vielfach vorkommenden Wörter richtig erkannt und indexiert sein muß, um das Wiederfinden des enthaltenden Textes zu sichern. Schon bei Texten einer durchschnittlichen Länge von etwa 140 Wörtern zeigt dieses Phänomen Wirkung [6]. Bei sehr kurzen OCR-Texten, wie sie hier betrachtet werden, ist dies anders. Viele relevante Begriffe erscheinen lediglich einmal in einem Dokument. Indexierung und Retrieval müssen diesem Umstand gerecht werden.

Das Ergebnis einer Suche in den OCR-Texten soll etwa das in Abb. 2 gezeigte Aussehen haben.

Es gibt nur wenige frei verfügbare Programme zur fehlertoleranten Suche in Texten, darunter `glimpse` [2], das im wesentlichen auf `agrep` [3] basiert. Für große Mengen kurzer Texte scheint Glimpse jedoch wegen seiner langen Laufzeiten weniger geeignet. Publikationen über weitere Verfahren beziehen sich oft nicht auf das Information Retrieval oder es wird nicht oder nur beispielhaft auf Implementierungsfragen eingegangen. Experimentelle Arbeiten beziehen sich oftmals nur auf kleinere Mengen längerer Texte.

Abbildung 2: Ausgabe eines Retrievals

3 Indexierung und fehlertolerante Suche mit Trigrammen

Folgende Wortfehlertypen können bei der OCR-Texterkennung typischerweise unterschieden werden:

(1) Ein einzelnes Zeichen wird eingefügt.

(2) Ein einzelnes Zeichen wird gelöscht.

(3) Ein Zeichen wird falsch erkannt, z.B. wird „`e`“ als „`o`“ erkannt.

(4) Statt eines Zeichens werden zwei Zeichen erkannt, z.B. wird „`rn`“ aus „`m`“ erkannt.

(5) Statt zweier Zeichen wird nur eines erkannt, z.B. wird „`m`“ aus „`rn`“ erkannt.

Formal könnten die letzteren auf die ersteren zurückgeführt werden. In der Praxis zeigt sich aber, daß manche Fehlerarten häufiger, andere seltener auftreten. Man könnte daher den unterschiedlichen Fehlerarten Gewichtungen zuordnen. Dies unterlassen wir aus Laufzeitgründen jedoch. Festzustellen ist jedenfalls, daß sich die mittels OCR erkannten Wörter um einzelne Zeichen von den vorgelegten Wörtern unterscheiden und im Fall (1) durch das Einfügen von Leerzeichen ein Wort durchaus in mehrere Wörter zerlegt werden kann.

Zur Indexierung werden zunächst die in den OCR-Texten enthaltenen Wörter bestimmt. Wörter sind aufeinanderfolgende Zeichenketten, die durch ein oder mehrere Interpunktionszeichen, Klammern oder Leerzeichen voneinander getrennt sind. Am Zeilenende getrennte Wörter werden zu zusammengesetzten Wörtern mit Bindestrich[4] zusammengefügt. Anschließend werden in den so erhaltenen Zeichenketten die Umlaute aufgelöst, Groß- in Kleinbuchstaben umgewandelt und sämtliche Zeichen, die nicht dem Alphabet „a“ ... „z“ oder den Ziffern „0“ ... „9“ entstammen, einheitlich in „_“ umgewandelt und an Wortgrenzen gleich wieder gelöscht. Schließlich werden die erhaltenen Wörter ggf. auf die ersten $c_{\max}$ Zeichen gekürzt und Wörter ignoriert, die aus weniger als $c_{\min}$ Zeichen bestehen. Wir setzen $c_{\min} = 3$ und $c_{\max} = 20$, was angesichts einer empirisch ermittelten durchschnittlichen Wortlänge von 9 Zeichen sicherlich gerechtfertigt werden kann.

Ein typischer Erkennungsfehler von OCR-Verfahren ist das Einfügen von Leerzeichen in Wörter. Um auch dies zu berücksichtigen, werden neben den Wörtern auch die Aneinanderkettung von Wortpaaren berücksichtigt. Vielfach vorkommende Wörter oder Wortpaare werden nur einmal berücksichtigt.

Als Ergebnis all dieser Operationen auf den OCR-Texten erhalten wir zu jedem Image I_i eine Wortmenge Ψ_i, die die verarbeiteten verschiedenen Wörter enthält. Analog wird verfahren, wenn mehrere Texterkennungsprogramme angewendet werden; in diesem Fall werden die Wortmengen eines I_i zu einem einzigen Ψ_i verschmolzen. Es gilt dann $\Psi_i = \{\psi_{i,0}, \ldots, \psi_{i,k_i-1}\}$ mit einem $k_i \geq 0$, wobei die $\psi_{i,j}$ Zeichenketten der Länge $c_{\min}$ bis $c_{\max}$ mit Zeichen aus { „_“, „a“, ..., „z“, „0“, ..., „9“} sind, an deren Anfang und Ende nur Buchstaben oder Ziffern vorkommen können.

Wesentlich für das Retrieval ist es, die in einer Query Φ vorkommenden φ_l mit den Wörtern $\psi_{i,j} \in \Psi_i$ zu vergleichen. Die Ψ_i sind daher Grundlage einer Datenbasis, die mit einer relationalen Datenbank verwaltet wird.

Zum Vergleich zweier Wörter werden Trigramme herangezogen. Trigramme sind die verschiedenen Buchstabentripel, aus denen ein Wort besteht. Zur angemessenen Berücksichtigung der Wortgrenzen wird an Wortanfang und -ende je zwei zuätzliche Zeichen „__“ einbezogen. So

[4]Bei einem am Zeilenende getrennten Wort ist nicht entscheidbar, ob es sich um ein zusammengesetztes Wort mit Bindestrich handeln könnte oder nicht. Daher wird generell mit Bindestrich zusammengesetzt, der später in „_“ umgewandelt wird.

lauten die 10 Trigramme von `eberhard`: `__e`, `_eb`, `ebe`, `ber`, `erh`, `rha`, `har`, `ard`, `rd_`, `d__`.

Sei $\Psi_i = \{\psi_{i,0} \ldots \psi_{i,k_i-1}\}$ die Menge der zum Dokument K_i per OCR ermittelten verschiedenen Wörter. Zum Vergleich eines Wortes $\varphi_l \in \Phi$ einer Query mit einem Wort $\psi_{i,j} \in \Psi_i$ wird das Verhältnis $p(\varphi_l, \psi_{i,j})$ der Zahl der in beiden Wörtern übereinstimmenden Trigramme zur Zahl der in φ_l vorkommenden Trigramme bestimmt. Ist beispielsweise $\varphi_l =$ `eberhard` und $\psi_{i,j} =$ `eborhard` mit den Trigrammen `__e`, `_eb`, `ebo`, `bor`, `orh`, `rha`, `har`, `ard`, `rd_`, `d__`, stimmen hiervon 7 Trigramme mit denen von φ_l überein, so daß $p(\varphi_l, \psi_{i,j}) = 0.7$ folgt.

Aus $\varphi = \psi$ folgt $p(\varphi, \psi) = 1$. Die Umkehrung gilt jedoch nicht, wie das Beispiel $\varphi =$ `schrift` und $\psi =$ `schoenschrift` zeigt: Sämtliche Trigramme von φ kommen auch in ψ vor, so daß $p(\varphi, \psi) = 1$ gilt. Um dies auszugleichen und mit Blick auf das Antwortzeitverhalten soll das Vergleichsverfahren leicht erweitert werden. Der Vergleich eines φ_l mit einem $\psi_{i,j}$ soll unterlassen werden, wenn $\psi_{i,j}$ viel länger oder viel kürzer als φ_l ist. Dazu sei $len(\omega)$ die Länge des Wortes ω, d.h. die Anzahl seiner Zeichen, und $\varepsilon_1, 0 \leq \varepsilon_1 \leq 1$, ein Parameter, etwa $\varepsilon_1 = 0.3$. Mit

$$\lambda(\varphi_l) := \begin{cases} \lfloor \varepsilon_1 \cdot len(\varphi_l) + 0.5 \rfloor & \text{, falls } \; len(\varphi_l) > c_{\min} \\ 0 & \text{, sonst} \end{cases}$$

sowie

$$\begin{aligned} \lambda_{\min}(\varphi_l) &:= \min\{c_{\min}, len(\varphi_l) - \lambda(\varphi_l)\} \\ \lambda_{\max}(\varphi_l) &:= \max\{c_{\max}, len(\varphi_l) + \lambda(\varphi_l)\} \end{aligned}$$

sei $\Psi'_i := \{\psi_{i,j} \in \Psi_i \mid \lambda_{\min}(\varphi_l) \leq len(\psi_{i,j}) \leq \lambda_{\max}(\varphi_l)\}$. Ψ'_i enthalte also ausschließlich Wörter aus Ψ_i, deren Länge sich, abhängig vom Parameter ε_1, nur wenig von $len(\varphi_l)$ unterscheidet. Für $\varepsilon_1 = 0.3$, $c_{\min} = 3$ und $c_{\max} = 20$ ergeben sich beispielsweise folgende Werte:

$len(\varphi_l)$	$\lambda(\varphi_l)$	$\lambda_{\min}(\varphi_l)$	$\lambda_{\max}(\varphi_l)$	$len(\varphi_l)$	$\lambda(\varphi_l)$	$\lambda_{\min}(\varphi_l)$	$\lambda_{\max}(\varphi_l)$	$len(\varphi_l)$	$\lambda(\varphi_l)$	$\lambda_{\min}(\varphi_l)$	$\lambda_{\max}(\varphi_l)$
3	0	3	3	9	3	6	12	15	5	10	20
4	1	3	5	10	3	7	13	16	5	11	20
5	2	3	7	11	3	8	14	17	5	12	20
6	2	4	8	12	4	8	16	18	5	13	20
7	2	5	9	13	4	9	17	19	6	13	20
8	2	6	10	14	4	10	18	20	6	14	20

Hier ist ein Beispiel angebracht: Ist $\varphi_l =$ `eberhard`, so ist $\lambda_{\min}(\varphi_l) = 6$ und $\lambda_{\max}(\varphi_l) = 10$. Die Query `eberhard` wird daher nur mit solchen Wörtern aus Ψ_i verglichen, die zwischen 6 und 10 Zeichen lang sind. Alle anderen Wörter von Ψ_i werden beim Vergleich ignoriert, weil sie als zu verschieden von der Query angesehen werden.

Wir setzen nun

$$p'(\varphi_l, \psi_{i,j}) := \begin{cases} p(\varphi_l, \psi_{i,j}) & \text{, falls } \psi_{i,j} \in \Psi'_i \\ 0 & \text{, sonst.} \end{cases}$$

Wird p' statt p als Vergleichsoperation verwendet, verkleinert sich die Menge der Wörter, deren Trigramme zum Vergleich herangezogen werden, durch Längenbeschränkung. Damit wollen wir uns aber noch nicht zufrieden geben. Auch die Menge der zu berücksichtigenden Ψ'_i soll verringert werden: Solche Ψ'_i, die nur Wörter geringer Trigrammähnlichkeit zu den Wörtern der Query enthalten, sollen beim Ranking ebenfalls unberücksichtigt bleiben. Dazu wird ein zweiter Parameter ε_2, $0 \leq \varepsilon_2 \leq 1$, eingeführt. Damit sei

$$p''(\varphi_l, \psi_{i,j}) := \begin{cases} p'(\varphi_l, \psi_{i,j}) & \text{, falls } p'(\varphi_l, \psi_{i,j}) \geq 1 - \varepsilon_2 \\ 0 & \text{, sonst.} \end{cases} \tag{1}$$

In der Praxis ergab sich $\varepsilon_2 = 0.5$ als akzeptabler Parameter[5]. Durch Verwendung von p'' statt p' als Vergleichsoperation verkleinert sich die beim Ranking zu berücksichtigenden Ψ'_i drastisch. Ist beispielsweise $\varphi_l =$ `eberhard`, so bleiben Wörter, die so geringe Ähnlichkeit damit haben wie `heidelberg` oder `erhebung` mitsamt der sie enthaltenden Ψ'_i ohne Berücksichtigung, falls diese Ψ'_i nicht noch ähnlichere Wörter zu φ_l enthalten. Auch $\psi_{i,j} =$ `reinhard`, für das $p(\varphi_l, \psi_{i,j}) = 0.4$ gilt, bliebe bei $\varepsilon_2 = 0.5$ im Ergebnis unberücksichtigt mit der Folge, daß auch das enthaltende Ψ'_i nicht in das Ranking einfließt.

Den Wert

$$p_{l,i} := \max_{0 \leq j \leq k_i - 1} \{p''(\varphi_l, \psi_{i,j}) \mid \psi_{i,j} \in \Psi'_i\} \tag{2}$$

nehmen wir als Maß dafür, daß K_i das Wort φ_l enthält. Es gilt $0 \leq p_{l,i} \leq 1$. Ein nicht verschwindendes $p_{l,i}$ wird also von einem Wort aus Ψ'_i bestimmt, das im Trigrammvergleich die größte Ähnlichkeit zu φ_l aller Wörter aus Ψ'_i aufweist und diesem außerdem ähnlich genug ist. Ist beispielsweise wieder $\varphi_l =$ `eberhard` und enthält ein Ψ'_i die beiden Wörter $\psi_{i,j} =$`reinhard` und $\psi_{i,k} =$`eborhard` und sonst nur Wörter geringerer Ähnlichkeit zu φ_l, so folgt mit den oben definierten ε_1 und ε_2: $p_{l,i} = 0.7$. Im Gegensatz zu [4] werden also die Ähnlichkeitsmaße mehrerer Wörter aus Ψ'_i nicht miteinander kombiniert.

Der Anwenderin werden die zu ihrer Query Φ gefundenen K_i in der Reihenfolge ausgegeben, die von der Funktion $rsv(\Phi, K_i)$ bestimmt wird. Der Ansatz zur Bestimmung von rsv, dem *retrieval status value*, basiert auf den $p_{l,i}$ aus (2). Er folgt ansonsten der Herleitung in [4]:

$$rsv(\Phi, K_i) := \sum_{\phi_l \in \Phi} a_{l,i} b_l \tag{3}$$

mit

$$a_{l,i} := p_{l,i} \quad \text{und} \quad b_l := \log \frac{1+n}{1 + \sum_{j=0}^{n-1} p_{l,j}} \quad , \tag{4}$$

[5]Der Benutzerin könnte überlassen werden, die Parameter ε_1 und ε_2 maßvoll zu variieren.

wobei n die Gesamtzahl der Dokumente ist.

Im nachfolgenden Abschnitt wird die Architektur des Systems beschrieben, und im Anhang werden Kernelemente des Algorithmus' herausgearbeitet.

4 Architektur

Bei der Architektur wird von folgender Vorgehensweise der Benutzerin ausgegangen:

- Stellen einer Suchanfrage (Query).
- Betrachten eines Teils des Ergebnisses dieser Query, das im wesentlichen aus einer Teilliste von Images besteht (Abb. 2), beginnend mit den besten Treffern. Die Größe der Treffermengen macht ein Vorgehen in Portionen sinnvoll.
- Evtl. Browsen in weiteren Teilen der Imageliste. Dazu werden Navigationshilfen in Form von Links bereitgestellt.
- Ggf. Stellen einer neuen Query.

Die Vorgänge vom Stellen einer Query bis zum Stellen der nächsten Query werden unter dem Begriff *Session* zusammengefaßt. Mit jeder neuen Query beginnt eine neue Session. Auf Serverseite werden die Aufgaben im Zusammenhang mit der Query von denen für das Browsen in der gefundenen Treffermenge getrennt. Darüber hinaus werden beim Bearbeiten der Session zwei sequentiell durchzuführende Schritte unterschieden:

1. Finden aller Nummern i, für die Ψ'_i sämtliche Wörter der Query enthalten. Dies sind die besten Treffer. Dieser Vorgang wird als *Suchen* bezeichnet.
2. Finden der Nummern i solcher Ψ'_i, für die der retrieval status value gemäß (3) geringer ausfällt. Dieser Vorgang wird als *Ranking* bezeichnet.

Die Treffermenge des ersten Schrittes wird mit dem zweiten Schritt erneut ermittelt. Formal betrachtet ist der erste Schritt daher obsolet. Er dient lediglich der Minimierung der Reaktionszeit auf eine Suchanfrage, denn die Antwortzeit beim ersten Schritt ist vernachlässigbar (s. dazu Anhang). Die Dauer der Benutzerinteraktion auf Basis der Ergebnisse des ersten Schrittes wird aufgewendet, um den zweiten Schritt zu bearbeiten. Die Benutzerinteraktion wird also genutzt, um im Hintergrund die Berechnungen für Schritt zwei durchzuführen.

Die Implementierung jeder der beiden Schritte wird im Anhang skizziert. In diesem Abschnitt steht die Gesamtarchitektur im Vordergrund.

Die Aufgaben Suchen, Ranking und Browsen werden jeweils verschiedenen Prozessen zugeordnet (s. Abb. 3).

Die Query wird via Web-Server (Httpd) beim *Query Client* gestellt. Dieser nimmt die Suchanfrage entgegen und erteilt zuerst dem *Search Server* einen Suchauftrag gemäß Schritt 1.

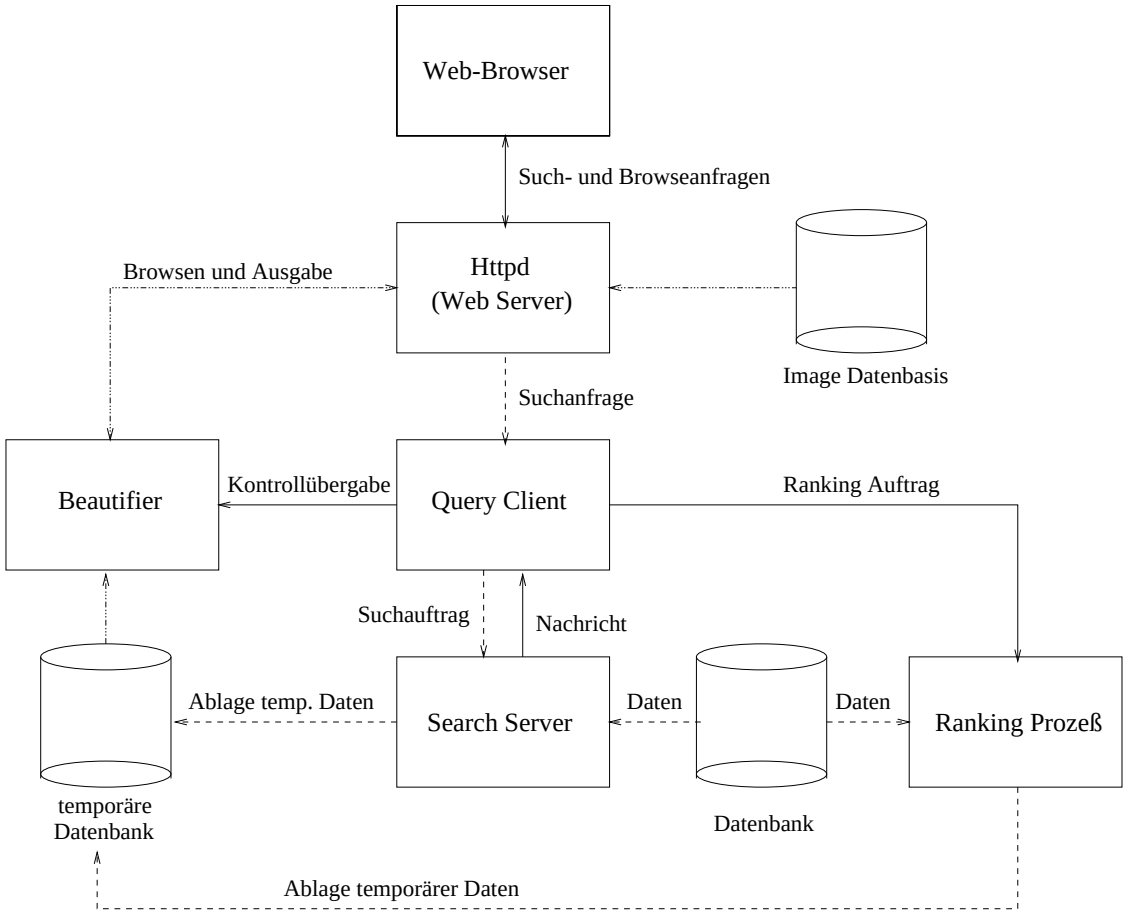

Abbildung 3: Jede Aufgabe ist spezifischen Prozessen des Systems zugeordnet.

Der Query Client wartet dann auf eine Nachricht des Search Servers.

Der Search Server wartet als Daemon resident auf Suchaufträge, wobei Latenzzeiten für den Programmstart praktisch entfallen. Der Search Server erhält seine Aufträge vom Query Client und bearbeitet die Suche gemäß Schritt 1. Seine Ergebnisse gibt er in eine temporäre Datenbanktabelle aus, die ausschließlich für diese Session eingerichtet wird. Nach der sehr kurzen Verarbeitungzeit des Search Servers wird der Query Client benachrichtigt. Dieser hat daraufhin noch zwei Aufgaben: zum einen erteilt er dem *Ranking Prozeß* einen Auftrag zur Bearbeitung des Schrittes 2, und zum zweiten gibt er die Kontrolle an den *Beautifier* ab. Der Query Client wartet die Ergebnisse des Ranking Prozesses nicht ab.

Der Beautifier dient der Benutzerinteraktion für das Browsen in der gefundenen Treffermenge einer Session. Er holt eine Portion aus der temporären Datenbanktabelle, bereitet sie für ein HTML-Dokument auf und gibt sie via Httpd an die Benutzerin weiter. Die Ausgaben des Beautifiers können etwa wie in Abb. 2 aussehen. Der Beautifier stellt Navigationshilfen in

Form von Links bereit, um der Benutzerin das Blättern in der Treffermenge zu erleichtern.

Während die Benutzerin die Ergebnisse des Schrittes 1 betrachtet und beurteilt, bearbeitet der Ranking Prozeß den aufwendigeren Schritt 2 und gibt schließlich auch dessen Ergebnisse in die vorhandene temporäre Datenbanktabelle aus. In der Tabelle wird abschließend vermerkt, daß der Ranking Prozeß für diese Query terminiert ist und keine weiteren Daten folgen. Der Ranking Prozeß wird vom Query Client als Hintergrundprozeß gestartet. Dies dient der Entkopplung von Query Client und Ranking Prozeß: Der Query Client brauch die Terminierung des Ranking Prozesses nicht abzuwarten.

Der Benutzerkommunikation zum Browsen in den Ergebnissen des Ranking Prozesses dient wieder der Beautifier. Erst eine neue Query bringt mit einer neuen Session wieder den Query Client, den Search Server und den Ranking Prozeß ins Spiel.

Bei dieser Architektur können Suchaufträge parallel bearbeitet werden. Werden mehrere Suchanfragen fast gleichzeitig abgesetzt, ist die Antwortzeit sämtlicher Suchanfragen gemäß Schritt 1 kaum spürbar, während die Rankingaufträge im Hintergrund verarbeitet werden und von evtl. mehreren vorhandenen Prozessoren profitieren. Darüber hinaus können die Programme für die Benutzerkommunikation schlank gehalten werden. Die temporäre Datenbanktabelle der Session wird nach einiger Zeit gelöscht.

5 Ausblick

Das in diesem Papier vorgestellte Verfahren zum fehlertoleranten Recherchieren in großen Mengen sehr kurzer heterogener OCR-Texte stellt eine Alternative zur manuellen Neuerfassung der Texte dar. Die Qualität der Retrievalergebnisse muß wegen der OCR-Texterkennungsfehler gegenüber der mit Neuerfassung erzielbaren Qualität sicherlich zurückstehen. Der enorme zeitliche und pekuniäre Aufwand für eine Neuerfassung wird jedoch vermieden.

Das Verfahren folgt Ideen in [4], die mit Blick auf einfache Implementierung und kurze Antwortzeiten vereinfacht worden sind.

Vorhandene Imagekataloge könnten zur Verbesserung der Trefferausbeute um das beschriebene Verfahren erweitert werden. Das Verfahren kann auch eingesetzt werden, um mehrere ursprünglich separate Image- oder datenbankbasierte Kataloge mittels „Stichwortsuche“ miteinander zu koppeln. Aus Benutzersicht kann dies die Separation verschiedener Kataloge weitgehend aufheben.

Anhang: Der Algorithmus

Die praktische Durchführung des Verfahrens basiert auf einer relationalen Datenbank mit insgesamt $3 \cdot (c_{\max} - c_{\min} + 1)$ Tabellen, die Angaben zu den indexierten Wörtern enthalten. Für `$len = $cmin ... $cmax` werden diese angelegt durch die Datenbankoperationen.

```
CREATE TABLE WORDS_$len (
  word          CHAR ($len) NOT NULL PRIMARY KEY,
  word_id       MEDIUMINT UNSIGNED NOT NULL
)

CREATE TABLE GRAMS_$len (
  gram          CHAR (3) NOT NULL PRIMARY KEY,
  listlength    MEDIUMINT UNSIGNED NOT NULL,
  word_id_list  MEDIUMBLOB NOT NULL
)

CREATE TABLE CARDS_$len (
  word_id       MEDIUMINT UNSIGNED NOT NULL PRIMARY KEY,
  listlength    MEDIUMINT UNSIGNED NOT NULL,
  card_num_list MEDIUMBLOB NOT NULL
)
```

Jede Tabelle enthält ausschließlich Daten zu Wörtern einer bestimmten Länge `$len`. Mit der Tabelle `WORDS_$len` werden die Wörter der Länge `$len` aus der Vereinigungsmenge $\Psi = \bigcup_{0 \le i \le n-1} \Psi_i$ durch einen Schlüssel repräsentiert. Diese Schlüssel können numerisch behandelt werden.

Die Tabelle `GRAMS_$len` enthält für jedes Trigramm `gram` eine Liste `word_id_list`, in der die `word_id` solcher Wörter `word` von `WORDS_$len` enthalten sind, in denen `gram` vorkommt. Die Liste wird als Blob abgelegt, und in `listlength` wird ihre Länge gespeichert.

Die Tabelle `CARDS_$len` schließlich enthält zur jeder `word_id` die numerische Liste sämtlicher i, für die das zu `word_id` gehörige Wort `word` Element von Ψ_i ist. Auch diese Liste ist als Blob[6] abgelegt, wobei in `listlength` ihre Länge vermerkt ist.

Für das Retrieval werden die gesuchten Wörter zuerst den gleichen Normierungen (s. Seite 6) unterzogen, wie die Wörter in der Datenbank. Das Retrieval erfolgt dann in zwei Schritten:

1. Finden aller Nummern i, für die Ψ'_i sämtliche Wörter der Query enthalten. Dies sind die besten Treffer. Dieser Vorgang wird als *Suchen* bezeichnet.
2. Finden der Nummern i solcher Ψ'_i, für die der retrieval status value gemäß (3) geringer ausfällt. Dieser Vorgang wird als *Ranking* bezeichnet.

[6]Zur Implementierung haben wir die *Perl Data Language* verwendet. Die Listen `word_id_list` und `card_num_list` werden jeweils als Vektor (*piddle*) im Blob gespeichert. Die Vektoren könnten zuvor noch komprimiert werden, etwa mit `gzip` (siehe `http://www.cpan.org`), worauf wir hier jedoch aus Transparenzgründen verzichten.

Der erste Schritt dient ausschließlich der Verkürzung der Antwortzeiten. Er stellt sicher, daß Anfragen der Benutzerin praktisch ohne Verzögerung beantwortet werden können und die Bearbeitungsdauer der Benutzerin verwendet werden kann, um die komplexeren Vorgänge des zweiten Schrittes durchzuführen. Formal betrachtet ist der erste Schritt obsolet. Siehe auch Seite 9.

Der erste Schritt wird vom *Search Server* bearbeitet. Er ist mit wenigen Programmzeilen zu bewältigen, wenn die vorverarbeiteten Wörter der Query im Array `@query` stehen und die Ausgabe in `$card_list` erwartet wird:

```
$card_list = pdl null;
for $q ( @query ){
    $len = length $q;
    ( $word_id ) = $dbh->selectrow_array(
        "SELECT word_id FROM WORDS_$len WHERE word = \"$q\"" );

    $sth = $dbh->prepare(
        "SELECT listlength, card_num_list FROM CARDS_$len WHERE word_id = ?" );
    $sth->execute( $word_id );
    ( $listlength, $blob ) = $sth->fetchrow;

    # Append all piddles card_num_list into a single piddle $card_list
    $pdl = new PDL;
    $pdl->setdims( [$listlength] );
    $pdlref = $pdl->get_dataref();
    $$pdlref = $blob;
    $card_list = $card_list->append( $pdl )->sever;
}

# Compute numbers of cards containing all query terms using run length encoding
( $count, $values ) = $card_list->qsort->rle;
$card_list = $values->where( $count == scalar @query )->sever;
```

Für den zweiten Schritt, den der *Ranking Prozeß* bearbeitet, werden die Trigramme jedes Wortes `$q` von `@query` benötigt. Sie seien jeweils als Array in `@{$trigrams{$q}}` gespeichert. Auch die Werte $\lambda_{\min}$ und $\lambda_{\max}$ seien bereits bestimmt und in `$lambdamin{$q}` bzw. `$lambdamax{$q}` abgelegt. Der Parameter ε_2 sei in `$epsilon2` gespeichert. Im folgenden Programmstück werden für jedes `$q` die Werte p'' gemäß (1) bestimmt, und zwar zunächst noch unterschieden nach den verschiedenen Längen `$len = $lambdamin{$q} ... $lambdamax{$q}`. Besteht die Query aus m Wörtern, erhalten wir als Ergebnis insgesamt $2 \cdot m \cdot (\lambda_{\max} - \lambda_{\min} + 1)$ Vektoren: `$p2_word_values{$q}{$len}` enthält die Werte p'' gemäß (1) für sämtliche Wörter `word`, deren Schlüssel `word_id` im Vektor `$p2_word_indics{$q}{$len}` abgelegt sind.

```
for $q ( @query ){
    for $len ( $lambdamin{$q} ... $lambdamax{$q} ){
        $sth = $dbh->prepare(
            "SELECT listlength, word_id_list FROM GRAMS_$len WHERE gram = ?" );
```

```
        $index = zeroes 100000;
        $index_from = 0;

        for $t ( @{$trigrams{$q}} ){
            $sth->execute( $t );
            ( $listlength, $blob ) = $sth->fetchrow;

            # Append all piddles word_id_list into a single piddle $index
            $pdl = new PDL;
            $pdl->setdims( [$listlength] );
            $pdlref = $pdl->get_dataref();
            $$pdlref = $blob;

            $index_to = $index_from + $listlength - 1;
            # Get more memory for $index
            # This reduces the number of calls to the slow append function
            while ( $index_to >= nelem( $index )){
                $index = $index->append( $index->zeroes )->sever;
            }
            $slice = "$index_from:$index_to";
            ( $dummy = $index->slice( $slice )) .= $pdl;
            $index_from = $index_to + 1;
        }
        $index = $index->where( $index )->sever;

        # Sum up multiplicity of each word_id into piddle $values
        ( $step, $indics ) = $index->qsort->rle;
        $step = rld $step, $step->sequence;
        $values = zeroes( 1 + $step->at( -1 ));
        indadd 1 / ( scalar @{$trigrams{$q}} ), $step, $values;

        # Drop small values and corresponding indics
        $values = ( $values >= 1 - $epsilon2 );
        $p2_word_indics{$q}{$len} = $indics->where( $values )->sever;
        $p2_word_values{$q}{$len} = $values->where( $values )->sever;
    }
}
```

Nun gilt es, daraus $a_{l,i} = p_{l,i}$ und b_l gemäß (2) bzw. (4) zu bestimmen[7]. Dabei wird implizit vorgegangen, d.h. $a_{l,i}$ wird nicht direkt bestimmt, sondern gleich eine Vorstufe des *rsv* (3). Mit folgendem Programmstück werden zwei Vektoren berechnet: `$rsv_init_values` enthält Werte des *rsv* solcher Karten, deren Indizes in `$rsv_init_indics` stehen. Diese Vektoren enthalten evtl. noch Vielfachheiten, falls `@query` mehrere Suchworte enthält. Diese werden hier noch nicht aufgelöst.

```
$rsv_init_indics = pdl null;
$rsv_init_values = pdl null;
```

[7]Randfälle wie z.B. leere Vektoren seien überall der Transparenz halber außer acht gelassen.

```
$index_from = 0;
for $q ( @query ){
    $card_indics = zeroes 10000;
    $card_values = zeroes 10000;
    for $len ( $lambdamin{$q} ... $lambdamax{$q} ){
        $sth= $dbh->prepare(
            "SELECT listlength, card_indics FROM CARDS_$len WHERE word_id = ?" );

        for $i ( list $p2_word_indics{$q}{$len}->sequence ){
            $sth->execute( $p2_word_indics{$q}{$len}->at( $i ));
            ( $listlength, $blob ) = $sth->fetchrow;

            # Append all piddles card_indics into a single piddle $indics
            $indics = new PDL;
            $indics->setdims( [$listlength] );
            $pdlref = $indics->get_dataref();
            $$pdlref = $blob;
            # Set $values to corresponding word_id multiplicities
            $values = zeroes( $listlength ) + $p2_word_values{$q}{$len}->at( $i );

            $index_to = $index_from + $listlength - 1;
            # Get more memory for $card_indics and $card_values
            while ( $index_to >= nelem( $card_indics )){
                $card_indics = $card_indics->append( $card_indics->zeroes )->sever;
                $card_values = $card_values->append( $card_indics->zeroes )->sever;
            }

            # Store these $indics and $values into $card_indics and
            # $card_values, respectively
            $slice = "$index_from:$index_to";
            ( $dummy = $card_indics->slice( $slice )) .= $indics;
            ( $dummy = $card_values->slice( $slice )) .= $values;
            $index_from = $index_to + 1;
        }
    }
    $card_indics = $card_indics->where( $card_values )->sever;
    $card_values = $card_values->where( $card_values )->sever;

    # Sort according to increasing card numbers
    $index_temp = $card_indics->qsorti;
    $card_indics = $card_indics->index( $index_temp )->sever;
    $card_values = $card_values->index( $index_temp )->sever;

    # Compute multiplicities of $card_indics using run length encoding
    ( $temp, $indics ) = $card_indics->rle;
    $indics = $indics->where( $indics )->sever;

    $step = rld $temp, $temp->sequence;
    $prod = 1 + $step->at( -1 );
    $max_series_length = max $temp;
```

```
    # Sum up multiple values for each card num
    $values = zeroes( $max_series_length * $prod );
    $modul = $step->sequence % $max_series_length;
    indadd $card_values, $step * $max_series_length + $modul, $values;
    $values = maximum( reshape $values, $max_series_length, $prod );

    # Weight and append results for each $q into $rsv_init_indics and $rsv_init_values
    $b = log ( (1 + $n) / ( 1 + $values->sumover ));
    $rsv_init_indics = $rsv_init_indics->append( $indics )->sever;
    $rsv_init_values = $rsv_init_values->append( $values * $b )->sever;
}
```

Bei Mehrwortqueries enthält `@query` mehr als ein Element. In diesem Fall müssen noch evtl. Vielfachheiten aufgelöst werden, um rsv gemäß (3) abschließend zu bestimmen. Darüber hinaus soll noch so normiert werden, daß Rangwerte zwischen 0 (schlechteste Treffer) und 100 (beste Treffer) entstehen. Als Ergebnis erhalten wir zwei Vektoren: `$rsv_values` enhält die Werte des rsv für Indizes, die in `$rsv_indics` gespeichert sind.

```
if ( scalar @query > 1 ){
    # Sort according to increasing card numbers
    $index_temp = $rsv_init_indics->qsorti;
    $rsv_init_indics = $rsv_init_indics->index( $index_temp )->sever;
    $rsv_init_values = $rsv_init_values->index( $index_temp )->sever;

    # Compute multiplicities of $rsv_init_indics using run length encoding
    ( $step, $rsv_indics ) = $rsv_init_indics->rle;
    $rsv_indics = $rsv_indics->where( $rsv_indics )->sever;

    # Sum up multiple $rsv_init_values into piddle $rsv_values
    $step = rld $step, $step->sequence;
    $rsv_values = zeroes( 1 + $step->at ( -1 ));
    indadd $rsv_init_values, $step, $rsv_values;
} else {
    # Handle one word queries
    $rsv_indics = $rsv_init_indics;
    $rsv_values = $rsv_init_values;
}

# Norm and drop least relevant $rsv_values
$rsv_values = rint( $rsv_values * ( 100 / $rsv_values->max ))
$index_temp = $rsv_values >= 100 * ( 1 - $epsilon2 );
$rsv_indics = $rsv_indics->where( $index_temp )->sever;
$rsv_values = $rsv_values->where( $index_temp )->sever;
```

Die Ergebnisvektoren `$rsv_values` und `$rsv_indics` sind jetzt noch geeignet aufzubereiten und auszugeben. Gemäß Abschnitt 4 geschieht dies durch Abspeichern der Ergebnisvektoren in eine temporäre Datenbanktabelle. Der Beautifier, auf dessen Implementierung nicht eingegangen werden soll, übernimmt diese Daten schließlich für die Web-Präsentation.

Literatur

[1] C. Fabian, K. Haller *Der Image-Katalog als alternatives Mittel der Konversion*, ZfBB 45(1998), Heft 2

[2] Udi Manber, Sun Wu *GLIMPSE: A Tool to Search Through Entire File Systems*, Technical Report 93-34, Department of Computer Science, University of Arizona, Tucson, Arizona

[3] Udi Manber, Sun Wu *Agrep: A Fast Approximate Pattern-Maching Tool*, Usenix Winter 1992 Technical Conference, San Francisco (Januar 1992), pp. 153 – 162

[4] Elke Mittendorf, Peter Schäuble, Páraic Scheridan *Applying Probabilistic Term Weighting to OCR Text in the Case of a Large Alphabetic Library Catalogue*, in: Fox, Ingwersen, Fidel: SIGIR 95 Procedings of the 18th Annual International ACM SIGIR Conference, Seattle 1995, Special Issue of the SIGIR Forum

[5] Eberhard Pietzsch *Kostengünstige Digitalisierung eines Zettelkataloges*, in: Zeitschrift für Bibliothekswesen und Bibliographie, 45 (1998), pp. 479-494

[6] Kazem Taghva, Julie Borsack, Allen Condit, Padma Inaparthy *Querying Short OCR'd Documents*, Technical Report 94-10, Information Science Research Institute, University of Nevada, Las Vegas