Wirtschaftsuniversität Wien
Vienna University of Economics and Business Administration


# Bachelorarbeit

| | |
|---|---|
| **Titel der Bachelorarbeit:** | **Full Text Search in Databases** |
| **Verfasser/in** (Nachname, Vorname): | **Kainrath Stefan** |
| **Matrikelnummer:** | **0651066** |
| **Studium:** | **Bawiso06** |
| **Titel der Lehrveranstaltung:** (in deren Rahmen die Bachelorarbeit abgefasst wurde) | **Datenbanksysteme** |
| **Beurteiler/in** (Titel, Vorname, Nachname): | **Dipl.-Ing. Mag. Dr. Albert Weichselbraun** |

Hiermit versichere ich, dass

1.) ich die vorliegende Bachelorarbeit selbständig und ohne Verwendung unerlaubter Hilfsmittel verfasst habe. Alle Inhalte, die direkt oder indirekt aus fremden Quellen entnommen sind, sind durch entsprechende Quellenangaben gekennzeichnet.

2.) die vorliegende Arbeit bisher weder im In- noch im Ausland zur Beurteilung vorgelegt bzw. veröffentlicht worden ist.

3.) diese Arbeit mit der beurteilten bzw. in elektronischer Form eingereichten Bachelorarbeit übereinstimmt.

4.) (nur bei Gruppenarbeiten): die vorliegende Arbeit gemeinsam mit (Vorname, Nachname) entstanden ist. Die Teilleistungen der einzelnen Personen sind kenntlich gemacht, ebenso wie jene Passagen, die gemeinsam erarbeitet wurden.

_____ _____

Datum Unterschrift

## Abstrakt

In heutigen Informationssystemen ist es von großer Wichtigkeit, schnell und einfach an die gewünschten Informationen zu gelangen. Performante und zuverlässige Volltextsuchen sind daher für viele Anwendungsfälle von hoher Bedeutung. Inhalt dieser Arbeit ist zunächst eine theoretische Abhandlung über Volltextsuche. Im Anschluss werden die Frameworks Tsearch2 und Lucene vorgestellt. Des Weiteren werden zwei Suchalgorithmen, die einen Invertierten Index verwenden, in PostgreSQL implementiert. Bei der zweiteiligen Suche wird zuerst die verwendete Tabelle indiziert. Führt der Anwender eine Suche aus, so werden die zuvor aufbereiteten Daten durchsucht und die Ergebnisse nach der Relevanz sortiert. Zu guter Letzt werden die erzielten Ergebnisse im Hinblick auf Funktionalität und Performance getestet.



## Abstract

In current information systems it is of prime importance to obtain the desired information quick and without complexity. Therefore reliable and high performance full text search algorithms are often a significant aspect in applications. Content of this thesis is a short disquisition of general literature about full text search, followed by an introduction of the frameworks Tsearch2 and Lucene. Furthermore, two search algorithms, which use an inverted index, are implemented in PostgreSQL. The search progress is split into two parts. At first the used table is indexed. When the user executes a query the previously processed data get scanned. Thereby the results are sorted by relevance. Last but not least the achieved results are tested with regard to functionality and performance.



## Keywords

Full Text Search, Tsearch2, Lucene, PostgreSQL, Python, Reverse Index

# Inhaltsverzeichnis

# 1 Einleitung

Ziel dieser Arbeit, die im Rahmen des Bachelorstudiums Wirtschaftsinformatik an der WU-Wien verfasst wurde, ist es, einen Einblick in das Thema Volltextsuche in Datenbanken zu geben und anderen Kollegen eine gute Einstiegsmöglichkeit zu diesem Thema zu verschaffen.

## 1.1 Problemstellung und Motivation

Im heutigen Informationszeitalter trifft man immer öfters auf Softwaresysteme, welche eine Vielzahl an Informationen sammeln und den Anwendern bereitstellen. Meist werden diese Datenbestände in Datenbanken abgespeichert. Es ist für die Anwender dieser Informationssysteme von großer Wichtigkeit, schnell und einfach an die gewünschten Informationen zu gelangen.

Das ist auch bei dem System „Media Watch on Climate Change“ der Fall, welches Artikel mit Umweltschwerpunkt von 150 angloamerikanischen Nachrichtenseiten in wöchentlichen Intervallen visualisiert. Dabei werden von 300.000 Artikeln etwa 10.000 Artikel, welche über Umwelteinflüsse berichten, erfasst, bevor diese in einem zentralen Archiv gespeichert werden. Eine effiziente Volltextsuche und Rankingalgorithmen spielen hier eine entscheidende Rolle für eine benutzerfreundliche Schnittstelle.

In aktuellen PostgreSQL-Datenbanken gibt es verschiedene Wege, um eine Suche nach den gewünschten Daten zu implementieren. Liefert eine sequenzielle Suche bei kleinen Datenbeständen noch zufriedenstellende Ergebnisse, kann dieses Verfahren jedoch bei großen Datenmengen nicht mehr verwendet werden.

Ausgangspunkt dieser Arbeit ist eine simple Idee. Um nicht immer den gesamten Datenbestand durchsuchen zu müssen, wird eine Vorarbeit geleistet indem die Daten aufbereitet werden und in einem Index, welcher ein rasches Auffinden der benötigten Informationen ermöglicht, abgespeichert werden. Dieser enthält eine sortierte Ansammlung von Verweisen auf alle Informationen die durchsucht werden können. Beim Abfragen der Daten wird nicht mehr der gesamte Datenbestand durchsucht, sondern nur mehr die zuvor aufbereiteten Daten.

Inhalt dieser Arbeit ist es zunächst die Anforderungen an ein Suchsystem zu definieren. Der zweite theoretische Teil beschäftigt sich mit der Beschreibung der Architektur eines Suchsystems. Der erste praktische Abschnitt beschreibt die in PostgreSQL integrierte Suchfunktionalität Tsearch2. Im nächsten Teil wird die Open Source Lösung Lucene, mit der Volltextsuchen mit mehr als zufriedenstellenden Ergebnissen implementiert werden können, vorgestellt. Im Anschluss werden zwei in PostgreSQL implementierte Suchalgorithmen beschrieben. Zu guter Letzt werden die Leistungen mit denen der Tsearch2-Algorithmen verglichen.

## 1.2 Projektplan

Im nachfolgenden Balkendiagramm ist die Laufzeit und die zeitliche Anordnung der einzelnen Projektaufgaben ersichtlich.

Balkendiagramm (Gantt)
Vorgangsname
Dauer
Anfang
Ende
Vorgänger
Projektstart 0 Tage Mo 30.06.08 Mo 30.06.08
Projektmanagement 102 Tage? Mo 30.06.08 Fr 14.11.08
Projektabschluss 0 Tage Fr 14.11.08 Fr 14.11.08
Grobplanung 19 Tage? Mo 30.06.08 Di 22.07.08
Internetrecherche 15 Tage Mo 30.06.08 Mi 16.07.08
Literaturrecherche 5 Tage? Sa 05.07.08 Do 10.07.08
Arbeitsumgebung e 1 Tag? Fr 11.07.08 Fr 11.07.08
UML Modellierung 4 Tage Do 17.07.08 Di 22.07.08
Grobplanung abges 0 Tage Di 22.07.08 Di 22.07.08
Detailplanung 33 Tage Mi 23.07.08 Fr 05.09.08 4
Tsearch2 7 Tage Mi 23.07.08 Do 31.07.08
Lucene 7 Tage Fr 01.08.08 Mo 11.08.08 11
PPS 10 Tage Di 12.08.08 Mo 25.08.08 12
Spellchecking 4 Tage Di 26.08.08 Fr 29.08.08 13
Testcases definiere 5 Tage Mo 01.09.08 Fr 05.09.08 14
Detailplanung abges 0 Tage Fr 05.09.08 Fr 05.09.08 15
Umsetzung 28 Tage Mo 08.09.08 Mi 15.10.08 10
Tsearch2 7 Tage Mo 08.09.08 Di 16.09.08
Lucene 7 Tage Mi 17.09.08 Do 25.09.08 18
PPS 10 Tage Fr 26.09.08 Do 09.10.08 19
Spellchecking 4 Tage Fr 10.10.08 Mi 15.10.08 20
Umsetzung abgesc 0 Tage Mi 15.10.08 Mi 15.10.08 21
Performancemessung 11 Tage Do 16.10.08 Do 30.10.08 17
Test vorbereiten 5 Tage Do 16.10.08 Mi 22.10.08
Tsearch2 3 Tage Do 23.10.08 Mo 27.10.08 24
PPS 3 Tage Di 28.10.08 Do 30.10.08 25
Performancemessu 0 Tage Do 30.10.08 Do 30.10.08 26
Änderungen 10 Tage Fr 31.10.08 Do 13.11.08 23
16.06.
01 Juli
30.06.
14.07.
01 August
28.07.
11.08.
01 September
25.08.
08.09.
22.09.
01 Oktober
06.10.
20.10.
01 November
03.11.
17.11
30.06.
22.07.
05.09.
15.10.
30.10.
14.11.

# 2 Technischer Hintergrund

Im folgenden Abschnitt werden die eingesetzten Technologien im Detail beschrieben.

## 2.1 PostgreSQL

PostgresSQL ist ein leistungsfähiges, objektrelationales Datenbankmanagementsystem und steht unter der BSD Licence. PostgreSQL ist ein fortschrittliches Datenbanksystem, welches SQL92 und SQL99 unterstützt und viele Erweiterungen beinhaltet [Pos08b]. In dieser Arbeit wird mit der Version 8.3 gearbeitet.

### 2.1.1 Tsearch2

Bei Tsearch2 handelt es sich um eine in aktuellen PostgreSQL-Systemen integrierte Erweiterung zur Implementierung von Volltextsuchen. Diese erlaubt eine schnelle und performante Suche in Textfeldern.

### 2.1.2 Stored Procedures

Eine Stored Procedure bezeichnet eine gespeicherte Funktion in einem Datenbankmanagementsystem. Es können häufig verwendete Abläufe gespeichert werden, die dann auf dem System zur Verfügung stehen. Um diese Funktionen zu implementieren, können neben der Abfragesprache SQL auch weitere Befehle zur Ablaufsteuerung verwendet werden. In PostgreSQL ist es außerdem möglich weitere unterstützte Programmiersprachen wie Java und Python zu verwenden.

### 2.1.3 PL/pgSQL

PL/pgSQL (Procedural Language/PostgreSQL Structured Query Language) ist eine prozedurale Sprache, welche von PostgreSQL unterstützt wird. Sie erlaubt wesentlich komplexere Funktionen als SQL, wie z.B. Schleifen und andere Ablaufsteuerungsfunktionen.

### 2.1.4 Datenbanktrigger

Als Datenbanktrigger bezeichnet man eine Funktionalität von Datenbankmanagementsystemen die bei einer bestimmten Art von Operationen, wie z.B.

einer Änderung der Daten, eine Stored Procedure aufrufen. Dabei ist es möglich, die eigentliche Operation zu überprüfen, zu adaptieren und wenn nötig auch zu verhindern. Es können aber auch weitere Funktionen aufgerufen werden. Trigger werden deshalb meist zur Aufrechterhaltung der Integrität verwendet.

## 2.2 Python

Bei Python handelt es sich um eine Programmiersprache, welche die objektorientierte, aspektorientierte und auch die funktionale Programmierung unterstützt. In dieser Arbeit wird Python innerhalb von PostgreSQL für die Stored Procedures eingesetzt.

## 2.3 Lucene

Lucene ist ein Framework für Volltextsuchen. Als solches gibt es lediglich eine Grobstruktur vor und überlässt es dem Entwickler Detailaspekte zu implementieren. Lucene wurde gänzlich in Java geschrieben. Mittlerweile gibt es aber eine Vielzahl an Adaptierungen in anderen Programmiersprachen wie C, Perl und Python. In dieser Arbeit wird mit der Version 2.4.0 gearbeitet.

# 3 Anforderungen an das Suchsystem

In diesem Kapitel werden die Kriterien zur Bewertung eines Suchsystems behandelt. Zunächst werden allgemeine Kriterien beschrieben und im Anschluss befinden sich die Anforderungen für die zu implementierende Artikelsuche.

## 3.1 Allgemeine Anforderungen

In diesem Abschnitt werden allgemeine Kriterien für die Beurteilung eines Suchsystems aufgezählt [IRS00].

### 3.1.1 Performance

Für die meisten Anwender spielt es kaum eine Rolle wie viele Daten durchsucht werden müssen, so lange sie schnell und zielgerichtet auf die Daten zugreifen können. Ein wichtiges Kriterium für ein Suchsystem ist also die Wartezeit bis ein Ergebnis vorliegt, minimal zu halten. Aber nicht nur die Zeit für die Suche ist relevant, auch die Performance bei Vorbereitungsarbeiten (z.B. Anlegen des Index) und der Änderungsaufwand ist von Bedeutung. Weiters ist die Speichereffizienz ein wichtiges Kriterium. Da jedes System eine begrenzte Anzahl an Ressourcen hat, soll der Speicherbedarf gering gehalten werden.

### 3.1.2 Relevanz der Suchergebnisse

Neben den Performancemessungen gibt es auch noch eine nicht so leicht messbare qualitative Beurteilung. Darunter fällt zunächst die Relevanz der Resultate. So bringt es einem Anwender nur wenig, wenn zwar die Suche sehr schnell durchgeführt wird, jedoch die Treffer ohne Bedeutung sind. Die Relevanz eines Dokumentes unterliegt zwar einer subjektiven Bewertung, jedoch kann die Vollständigkeit (z.B. werden alle Dokumente mit einem bestimmten Inhalt gefunden) leichter überprüft werden. Bei der Bewertung der Güte eines Suchergebnisses haben sich die drei Maße Vollständigkeit (Recall), Genauigkeit (Precision) und Ausfallsquote (Fallout) durchgesetzt und lassen sich folgendermaßen berechnen [Ris09]:

$$\text{Recall} = \frac{\text{Menge der gefundenen relevanten Dokumente}}{\text{Menge der relevanten Dokumente in der Datenbank}}$$

$$\text{Precision} = \frac{\text{Menge der gefundenen relevanten Dokumente}}{\text{Menge der gefundenen Dokumente}}$$

$$\text{Fallout} = \frac{\text{Menge der gefundenen irrelevanten Dokumente}}{\text{Menge der irrelevanten Dokumente in der Datenbank}}$$

**Abbildung 1: Formeln zur Skalierbarkeit: Recall, Precision, Fallout**

### 3.1.3 Aktualität der Suchergebnisse

Es muss sichergestellt werden, dass auch geänderte und neue Daten gefunden werden. Gelöschte Daten hingegen dürfen nicht mehr gefunden werden.

### 3.1.4 Skalierbarkeit

Auch die Skalierbarkeit ist eine Anforderung an ein Suchsystem. So kann sich der Ressourcenbedarf bei der Erhöhung des Datenbestandes zwar erhöhen, sollte sich jedoch in einem akzeptablen Verhältnis befinden [lip06]. Ein wichtiger Begriff in diesem Zusammenhang ist die Speicher- oder Platzkomplexität. Darunter versteht man den minimalen Speicherbedarf eines Algorithmus zur Bewältigung eines Problems in Abhängigkeit zur verarbeiteten Menge der Daten. Ein ungünstiges Verhalten wäre demnach bei doppeltem Aufwand eine Verzehnfachung der benötigten Ressourcen.

### 3.1.5 Einschränkungen

Ein für den Anwender ebenfalls bedeutender Punkt ist die Möglichkeit, bei der Suche weitere Einschränkungen (z.B. durch Angabe der Quelle oder des Zeitpunktes der Erstellung des Datensatzes) angeben zu können.

### 3.1.6 Darstellung der Suchergebnisse

Zu guter Letzt ist die Aufbereitung der Daten für den Anwender wichtig. So können schon im Suchergebnis relevante Ausschnitte der gefundenen Daten angegeben werden.

## 3.2 Artikelsuche

In diesem Abschnitt werden nun die Bewertungskriterien für die zu implementierende Artikelsuche in PostgreSQL und der Suche mittels Tsearch-Framework definiert. Die allgemeinen Anforderungen sind größtenteils auch für die Artikelsuche relevant. Da aber die Implementierungen nicht direkt vom Anwender aufgerufen werden, kann auf die Punkte 3.1.5 und 3.1.6 verzichtet werden. In der hier vorliegenden Arbeit wird meist mit folgender Artikeltabelle gearbeitet:

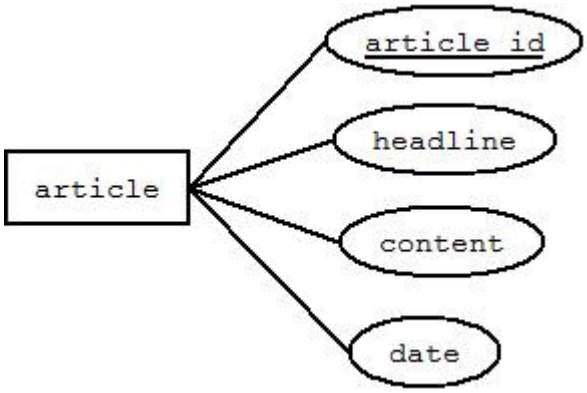


**Abbildung 2: ER: Artikeltabelle**

## 3.3 Hinweise zur Implementierung

Um den Lesefluss nicht zu stören befinden sich in der hier vorliegenden Arbeit nur Codebeispiele, um Funktionen und Abläufe besser zu verdeutlichen. Dabei wird keine Rücksicht auf eine Lauffähigkeit der einzelnen Teile genommen. Um den Code einfach zu halten, wird in den einzelnen Codebeispielen keine Fehlerüberprüfung durchgeführt bzw. werden keine Exceptions abgefangen. Der vollständig implementierte Code findet sich auf der beiliegenden CD wieder.

# 4 Architektur eines Suchsystems

Um eine gute Suchlösung zu erstellen, ist es von Nöten, die Abfragezeit zu minimieren. Eine einfache Idee ist, schon beim Anlegen eines Dokumentes dieses so vorzubereiten, sodass die nachfolgenden Suchabfragen schneller durchführen werden können. Um diesen Leitgedanken umsetzen zu können, ist es ratsam, die Suchapplikation in die Teile Informationsbeschaffung, Normalisierung, Indizierung und Suche aufzuspalten, welche im folgenden Abschnitt näher beschrieben werden[1].

[1] Weitere Informationen: Architektur eines Lucene-Suchsystems [Sol08]

## 4.1 Informationsbeschaffung

Um die Informationsbeschaffung durchzuführen, beinhalten viele Systeme einen sogenannten Crawler. Webcrawler sind in der Lage Webserver auf gewünschte Inhalte zu durchsuchen, so große Datenmengen zu finden und gewünschte Inhalte zu filtern. Eine weitere Art von Crawlern ist die Extraktion der Daten aus nicht textbasierten Dokumenten wie Microsoft Word-, PDF-, oder HTML-Dateien. Es gibt eigentlich keine relevanten, digitalen Informationsquellen, die nicht für eine Suche geeignet sind. Allerdings werden für diese eigene Schnittstellen benötigt. In dieser Arbeit wird allerdings nicht näher auf die Informationsbeschaffung eingegangen. Die Ausgangsbasis für die Suchen bildet ein Datenspeicher mit einer Vielzahl an Nachrichtentexten.

## 4.2 Normalisierung

Im nächsten Schritt können die gesammelten Rohdaten an einen Filter oder Analyser übergeben werden, welcher die Daten für die Indizierung aufbereitet. Diese Verarbeitung der Daten beschleunigt die späteren Suchabfragen, da die Datenmenge zum Teil beachtlich reduziert werden kann. Relativ leicht umzusetzen sind etwa die Normalisierung der Wörter in kleingeschriebene Terme und das Eliminieren der Stoppwörter[1]. Schwieriger zu realisieren sind linguistische Transformationen. So kann für jedes Wort der Stamm identifiziert werden und mit diesem weitergearbeitet werden. Dabei wird z.B. das Wort „suchen“ in „such“ umgewandelt. Diese Umwandlung ist allerdings an eine Vielzahl von Regeln verbunden und nicht immer einfach umzusetzen.

Um auch die gewünschten Informationen zu finden, muss natürlich auch die Abfrage mit demselben Analyser behandelt werden, der auch für die Erstellung des Index angewandt worden ist.

[1] Stoppwörter nennt man Wörter, die bei der Indizierung nicht beachtet werden, da sie sehr häufig auftreten und für das Ergebnis nicht relevant sind. Beispiele für üblicherweise verwendete Stoppwörter sind Artikel, Konjunktionen und Präpositionen [She01].

## 4.3 Indizierung

Natürlich ist es ist auch möglich, die vom Analyser gelieferten Daten zu durchsuchen, allerdings können die Suchen durch die richtige Anwendung eines Index wesentlich beschleunigt werden. Bei der Erstellung des Index hat der Entwickler wesentlichen Einfluss auf die Resultate, die später gefunden werden. So kann bestimmt werden, welche Daten in den Index aufgenommen werden. Dabei sollte allerdings beachtet werden, dass alle Informationen, die nicht im Index vorhanden sind auch nicht mehr im Suchergebnis enthalten sind. Die Qualität des Index determiniert also die Qualität des Suchergebnisses. Man muss sich schon zu Beginn darüber im Klaren sein, welche Informationen die Anwender wirklich benötigen, wie relevant einzelne Teilbereiche sind und nach welchen Kriterien gesucht werden kann.

Um einen Index abzuspeichern, kommen verschiedene Speicherarten in Frage. Im folgenden Abschnitt werden der Suchbaum und der Invertierte Index vorgestellt und miteinander verglichen.

### 4.3.1 Generalized Search Tree (GiST)

Ein GiST ist ein balancierter Suchbaum, für den es generische Methoden für das Suchen, Einfügen und Löschen der Daten gibt. Als Entwicklungsbasis dienten die verschiedensten B-Bäume, deren Funktionalitäten in einem einzigen Baum vereint werden sollten. Ein GiST ist eine erweiterbare Datenstruktur, die es erlaubt, jede Art von Daten zu indizieren und jede Art von Suchen ermöglicht.

Auch andere Suchbäume sind im Hinblick auf die handhabbaren Daten erweiterbar, jedoch sind sie in der Art der Abfragen eingeschränkt. So ermöglichen es B+-Bäume nur mit Vergleichsoperatoren (<, =, >) zu arbeiten und R-Bäume erlauben nur Bereichsabfragen. GiST-Indizes können zum einen diese Abfragen kombinieren zum anderen auch völlig neue Abfrageformen implementieren, die auf die verwendeten Datentypen zugeschnitten sind.

Der GiST hat eine ausbalancierte Baumstruktur, wie ein B-Baum. Er enthält ebenfalls Schlüssel-Wert-Paare, wobei ein Schlüssel aber nicht nur aus Zahlen bestehen kann, sondern

auch durch jede Art von Klassen repräsentieren werden kann. Diese können auch selbst definiert werden [BER99].

### 4.3.2 Generalized Invertierter Index

Wie auch schon bei einem GiST-Index bedeuted „generalized“, dass der GIN nicht auf einzelne Datentypen festgelegt ist, sondern dass es für beliebige Datentypen eigene Strategien gibt, um nach diesen zu suchen. [Sig06].

Ein invertierter Index ist ein Wort-orientierter Algorithmus, um Texte zu indizieren und dadurch die Suche zu beschleunigen. Dabei bezieht sich „invertiert“ auf die Tatsache, dass nicht die Dokumente auf die Wörter zeigen, sondern es existiert eine Liste von Wörtern die auf Dokumente verweisen, in denen diese vorkommen [wis09].

Ein Invertierter Index ist eine Datenstruktur, die aus einer Menge von (Schlüssel, Dokumentenliste)-Paaren besteht, wobei die Dokumentenliste wiederum eine Menge von Texten ist, in denen der Schlüssel vorkommt.

So ergeben die beiden Texte „Das ist ein GIN-Beispiel“ und „GIN verstehen mit Beispiel“ folgenden Invertierten Index:

```
das: 1
ist: 1
ein: 1
gin: 1,2
beispiel: 1,2
verstehen: 2
mit: 2
```

**Abbildung 3: Beispiel eines Invertierten Index**

Natürlich kann auch zusätzlich die Position mitgespeichert werden. So ergibt sich anstatt der Dokumentenliste eine Menge von (Dokument, Position)-Paaren. Bei demselben Beispiel ergibt sich folgender Index:

```
das: (1,1)
ist: (1,2)
ein: (1,3)
```

```
gin: (1,4),(2,1)
beispiel: (1,5),(2,4)
verstehen: (2,2)
mit: (2,3)
```

**Abbildung 4: Beispiel eines Invertierten Index (mit Position innerhalb des Textes)**

### 4.3.3 GiST und GIN in PostgreSQL

Ein GiST-Index in PostgreSQL kann mit `tsvectoren` und `tsqueries` arbeiten. Das sind zwei Datentypen, die für die Volltextsuche mit Tsearch benötigt werden (siehe 5.1 Tsearch2-Übersicht). Ein GIN-Index ist auf den Datentyp `tsvector` ausgelegt.

Es gibt aber substanzielle Geschwindigkeitsunterschiede. Der GiST-Index ist verlustbehaftet, da jedes Dokument durch Werte fixer Länge im Index repräsentiert wird, die aus den Hashes der einzelnen Wörter zusammengesetzt sind. Ergibt der Hashwert zweier Wörter zufälligerweise dieselbe Bitposition kann es zu falschen Treffern kommen. Dieser Fall muss durch eine zusätzliche Prüfung ausgeschlossen werden. PostgreSQL erledigt dies automatisch, es kommt aber dadurch zu Performanceeinbußen.

GIN-Indizes sind zwar nicht verlustbehaftet, die Performance hängt aber logarithmisch von der Anzahl der unterschiedlichen Wörter der Texte ab. Werden bei der Suche jedoch Gewichtungen einzelner Abfrageterme verwendet, so sind auch GIN-Indizes verlustbehaftet und müssen ebenfalls einer neuerlichen Prüfung unterzogen werden.

Laut PostgreSQL Dokumentation [Pos08a] gibt es vier grundlegende Unterschiede zwischen GIN- und GiST-Indizes:

- Abfragen bei einem GIN-Index sind etwa dreimal schneller als bei einem GiST-Index
- GIN-Indizes benötigen ungefähr dreimal so lange zum Erstellen als GiST-Indizes
- GIN-Indizes sind etwa zehnmal langsamer beim Updaten als GiST-Indizes
- GIN-Indizes sind ungefähr zwei bis dreimal größer als GiST-Indizes

Als Daumenregel kann gesagt werden, dass sich GIN-Indizes besser für statische Daten und GiST-Indizes eher für sich häufig ändernde Daten eignen. Je nach Anforderung muss deshalb unterschieden werden, welcher Index besser geeignet ist.

## 4.4 Suchen

Mit dem zuvor erstellten Index kann nun eine performante Suche durchgeführt werden. Dies kann in der Regel durch zwei verschiedene Arten geschehen. Erstens kann dem Anwender eine Auswahl verschiedenster Kriterien angeboten werden. Dies ist besonders bei unerfahrenen Benutzern ratsam und ist außerdem um ein Vielfaches schneller, da der Index auf die vorgegebene Art der Suchen optimiert werden kann. Die zweite Art von Suchen ist eine Volltextsuche mit einer speziellen Suchsyntax. Dabei kann im Datenbestand nach einzelnen Wörtern gesucht werden. Meist kann mittels einer eigenen Abfragesyntax eine komplexe Abfrage zusammengestellt werden. So können z.B. die einzelnen Abfrageterme mit Boolschen Operanden verknüpft oder Wildcards verwendet werden, um nur einige Möglichkeiten zu nennen.

# 5 Tsearch2

In diesem Kapitel wird das in PostgreSQL integrierte Tsearch2-Modul vorgestellt. Zuerst werden grundlegende Dinge wie die Datentypen `tsvetor` und `tsquery` vorgestellt und im Anschluss wird eine einfache Suchengine erstellt, um die schon zuvor erwähnte Artikeltabelle zu durchsuchen.

## 5.1 Übersicht

Zunächst stellt sich die Frage wozu man Tsearch2 benötigt. Um eine Suche zu implementieren bietet PostgreSQL schon seit Jahren die Suchoperatoren ~, ~*, LIKE und ILIKE. Diese Operatoren haben aber einige wesentliche Nachteile:

- Bei jeder Suche muss immer der gesamte Datenbestand durchsucht werden und deshalb sind diese Operatoren sehr langsam

- Es gibt keine Unterstützung für linguistische Mechanismen.
- Des Weiteren gibt es keine Möglichkeit die Ergebnisse nach der Relevanz zu sortieren

Diese Punkte werden durch die Volltextsuche mit Tsearch2 unterstützt. Um eine Suche durchführen zu können, ist es notwendig zuerst einen Index anzulegen. Zunächst muss ein Dokument in den Datentyp `tsvector` umgewandelt werden. Als `tsvector` bezeichnet man eine sortierte Liste aller im ursprünglichen Text vorkommenden normalisierten Wörter. Um diesen Schritt nur einmal durchführen zu müssen, ist es notwendig den `tsvector` abzuspeichern [Bar08].

Nach erfolgter Indizierung der Dokumente kann auch schon mit der Suche begonnen werden. Dabei muss die Suchabfrage in den Datentyp `tsquery` umgewandelt werden, was natürlich mit denselben Normalisierungsschritten wie das zu durchsuchende Dokument selbst verarbeitet werden muss. Nach dieser Umwandlung kann die eigentliche Suche durchgeführt werden. Zu guter Letzt bietet Tsearch2 die Möglichkeit die gefundenen Treffer aufzubereiten um den Informationsgrad zu erhöhen.

## 5.2 Indizierung

In diesem Abschnitt wird gezeigt, wie man einen `tsvector` anlegt und zur Wiederverwendung abspeichert.

### 5.2.1 Erstellung des `tsvectors`

Um einen `tsvector` anzulegen, wird zunächst ein Dokument in einzelne Token[1] aufgeteilt, welche in einem zweiten Schritt in sogenannte Lexeme umgewandelt werden. Ein Lexem ist ein normalisierter Token, wobei hier verschiedenste Normalisierungsmaßnahmen durchgeführt werden können (siehe Kapitel 4.2 Normalisierung).

Um aus einem Rohtext einen `tsvector` anzulegen stellt PostgreSQL die Funktion `to_tsvector(text)` zur Verfügung:

[1] Ein Token ist eine lexikalische Grundeinheit wie z.B. ein Wort, eine Zahl oder eine Emailadresse, welche von einem Parser verwendet wird [neo08].

```
SELECT to_tsvector('Tsvector Beispiel: Wie sieht ein tsvector aus?')
AS vector;

vector
----------
"'aus':7 'ein':5 'wie':3 'sieht':4 'beispiel':2 'tsvector':1,6"
```

**Listing 5-1: Tsearch2 – Erstellung eines tsvectors**

Auf den ersten Blick sieht das gelieferte Ergebnis korrekt aus. Es werden für alle verschiedenen Wörter die Positionen bestimmt und sortiert abgespeichert. Allerdings ist hier Vorsicht geboten. Bei Verwendung dieser Signatur der `to_tsvector`-Methode wird die Standardkonfiguration verwendet. PostgreSQL unterstützt derzeit verschiedenste Sprachen. Es kann hier sogar eine eigene Konfiguration erstellt werden. Dabei wird aber auf die Tsearch2-Reference verwiesen. Im folgenden Beispiel benötigen wir die deutsche Konfiguration:

```
SELECT to_tsvector('german', 'Tsvector Beispiel: Wie sieht ein
tsvector aus?') AS vector;

Vector
----------
"'sieht':4 'beispiel':2 'tsvector':1,6"
```

**Listing 5-2: Tsearch2 - Erstellung eines tsvectors2**

Bei diesem Beispiel ist zu erkennen, dass nicht mehr alle Wörter des ursprünglichen Textes im Ergebnis wiederzufinden sind. Der Grund liegt an den Stoppwörtern, die automatisch herausgefiltert werden.

Sehr nützlich für ein Suchsystem ist auch die Möglichkeit, mehrere Vektoren zusammenzuhängen. Sinn macht so eine Verknüpfung, wenn die einzelnen Vektoren unterschiedlich gewichtet werden. Dies geschieht mit der Funktion `setweight(vector, letter)`.

```
SELECT setweight(to_tsvector('german', 'Tsvector Beispiel: '), 'B')
 || to_tsvector('german', 'Wie sieht ein tsvector aus?') AS vector;
```

```
Vector
----------
"'sieht':4 'beispiel':2B 'tsvector':1B,6"
```

**Listing 5-3: Tsearch2: Vektoren gewichten und zusammenhängen**

### 5.2.2 Speichern des `tsvectors`

Um den `tsvector` nicht bei jeder Suchabfrage für jeden Datensatz neu zu erzeugen, sollte dieser abgespeichert werden. Eine Möglichkeit den Vektor abzuspeichern ist eine zusätzliche Spalte in der schon vorhandenen Datentabelle anzulegen. Aus Performancegründen sollte diese Spalte auch indiziert werden, da der Vektor nur einmal erstellt werden muss. Hier wird ein GIN verwendet (siehe Kapitel 4.3.2 Generalized Invertierter Index).

```
ALTER TABLE article
ADD COLUMN content_tsvector tsvector;

CREATE INDEX idx_article_content_tsvector
  ON article
  USING gin
  (content_tsvector);

UPDATE article
SET content_tsvector = to_tsvector('english', content)
                || setweight(to_tsvector('english', headline), 'C')
```

**Listing 5-4: Tsearch2 - Indizierung**

Zu beachten ist aber, dass der Index immer aktuell gehalten werden muss, da sonst die Suchergebnisse nicht mit den aktuellen Daten zusammenpassen. Um dies sicherzustellen wird ein Trigger erstellt, der bei jedem neuen bzw. geänderten Datensatz den Vektor neu bildet.

```
CREATE FUNCTION setTsvectorTrigger () RETURNS trigger AS
$$
    BEGIN
     NEW.content_tsvector :=
          to_tsvector('english', NEW.content)
          || setweight(to_tsvector('english', NEW.headline), 'C');
      RETURN NEW;
    END;
```

```
$$ LANGUAGE plpgsql;

CREATE TRIGGER tsvectorupdate BEFORE UPDATE OR INSERT ON article
            FOR EACH ROW EXECUTE PROCEDURE
            setTsvectorTrigger();
```

**Listing 5-5: Tsearch2 - Index aktuell halten**

## 5.3 Abfragen

Nachdem der Index angelegt wurde und sichergestellt ist, dass dieser zu jedem Zeitpunkt aktuell ist, kann auch schon mit der Suche begonnen werden.

### 5.3.1 Allgemein

Bei Tsearch2 werden die Suchabfrage mit dem Operator `@@` durchgeführt. Die Syntax sieht folgendermaßen aus:

```
tsvecotr @@ tsquery
```

**Listing 5-6: Tsearch2 –Allgemeine Abfragesysntax**

Als `tsvector` nehmen wir die indizierte Spalte `content_tsvector` aus dem vorhergehenden Kapitel. Die `tsquery` kann auf verschiedene Arten anlegen werden. Die erste Möglichkeit ist, sich die Abfrage selbst zusammenzubauen und auf eine `tsquery` zu casten:

```
SELECT article_id, headline
FROM artice
WHERE content_tsvector @@ 'python'::tsquery
```

**Listing 5-7: Tsearch2 - Abfrage mit Cast**

Diese Art hat aber den Nachteil, dass man mit den Abfragen sehr vorsichtig umgehen muss. Es gilt zu beachten, dass der Text des Artikels beim Einfügen umgewandelt wird. So würde eine Suche mit 'Python' zu keinem Ergebnis führen, da der gesamte Text in Kleinschreibung umgewandelt worden ist. Die für den Entwickler weitaus unproblematischere Variante ist die

Methode to_tsquery zu verwenden, bei der fakultativ die Konfiguration angegeben werden kann:

```
SELECT article_id, headline
FROM artice
WHERE content_tsvector @@ to_tsquery(‘default’, ’Python’);
```

**Listing 5-8: Tsearch2 - Abfrage mit to_tsquery**

Weiters können einzelne Wörter mittels Boolschen Operatoren verknüpft werden oder aber auch ganze `tsqueryies`. Bei diesen Operationen und weiteren Möglichkeiten sich eine `tsquery` anzulegen, verweise ich aber auf die Tsearch2-Referenz.

### 5.3.2 Ranking der Ergebnisse

Bis jetzt sind wir in der Lage alle relevanten Texte zu finden, jedoch wissen wir noch nicht, welcher Text am wertvollsten für uns ist. Für diesen Zweck gibt es in Tsearch2 die Methoden `ts_rank` und `ts_rank_cd`, die einen Vektor anhand einer Query aufgrund Nähe der einzelnen Wörter zueinander, der Länge des Dokuments und der Gewichtung der Terme. Je höher der Rang, desto besser der Vektor. Cd steht für Cover Density und bedeutet, dass die Position der Suchwörter im Text berücksichtigt wird.

```
CREATE TYPE tArticle AS (
     article_id integer ,
     headline character varying(512),
     date date,
     "content" text,
     content_tsvector tsvector,
     rank REAL
);


CREATE OR REPLACE FUNCTION findArticle(text)
RETURNS SETOF tArticle AS
$$
     SELECT article_id, headline, date, "content",
          content_tsvector, ts_rank(content_tsvector, query)
     FROM article, to_tsquery('english', $1) AS query
     WHERE content_tsvector @@ query
```

```
     ORDER BY ts_rank(content_tsvector, query) DESC
$$ LANGUAGE sql;

SELECT * FROM findArticle('europe');
```

**Listing 5-9: Tsearch2 – Abfragen auf die Artikeltabelle**

# 6 Apache Lucene

Lucene ist ein Projekt der Apache Software Foundation zur Unterstützung von Volltextsuchen, wobei nur der Kern der Suchengine zur Verfügung steht. Ohne die Detailaspekte anzubieten, bleibt Lucene sehr flexibel in der Anwendung [Luc08b]. In diesem Abschnitt werden zunächst die Bestandteile des Lucene-Frameworks vorgestellt und anschließend werden diese Teile in einer einfachen Java-Anwendung verwendet, um die Vor- und Nachteile von Lucene hervorzuheben.

## 6.1 Übersicht

Die Realisierung einer Volltextsuche in Lucene erfolgt in zwei Hauptarbeitsschritten. Zuerst wird der Datenbestand indiziert, um im Anschluss eine performante Suche durchführen zu können. Dabei werden die einzelnen Daten zunächst in Lucene-Dokumente umgewandelt. Im Anschluss werden diese von einem Analyser behandelt bevor diese im Index abgelegt werden.

Nachdem der Index erstellt wurde, kann eine Abfrage durchgeführt werden. Bei den Suchen muss selbstverständlich derselbe Analyser wie bei der Aufbereitung des Indexes verwendet werden. Folgende Abbildung erläutert das Zusammenspiel der Datenquelle bzw. der Suchabfrage mit Lucene [Sch06].

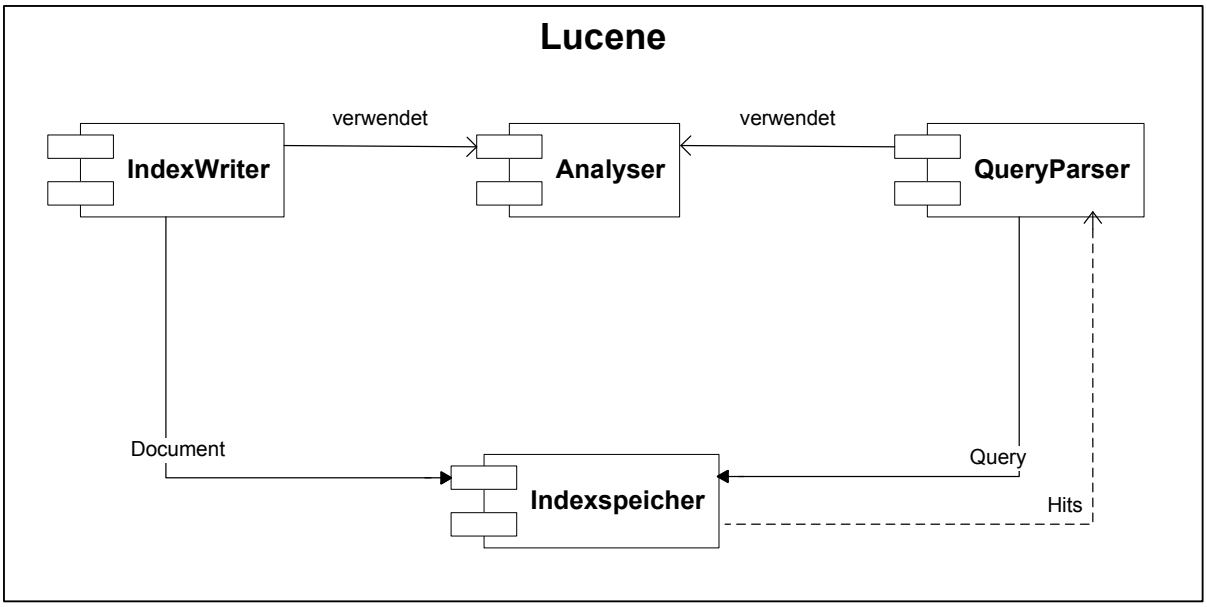


**Abbildung 5: Lucene – Ablauf einer Suche und das Zusammenspiel der Komponenten**

## 6.2 Indizierung

Der Index ist das Herzstück einer Suchapplikation. Im Prinzip kann mit Lucene jede Art von digitalen Textdaten abgespeichert und indiziert werden. Dabei muss jeder Datensatz in ein Lucene-Dokument umgewandelt werden. Ein Dokument besteht aus zumindest einem Feld, welches wiederum aus zwei Teilen, einem Namen und einem Wert, besteht. Das nachfolgende Codebeispiel zeigt die Indizierung von in einer Datenbank abgespeicherten Artikeln. Für eine genauere Dokumentation der verwendeten Klassen und Methoden verweise ich auf die offizielle Lucenedokumentation [Luc08a].

Zuerst wird ein `IndexWriter` angelegt. Als Parameter werden der Pfad zum Index, ein Analyzer und ein Boolscher Wert, welcher angibt, ob der Index neu angelegt wird oder ein bestehender erweitert wird, übergeben. Einfachheitshalber wird in dem Codebeispiel der `StandardAnalyzer` von Lucene verwendet. Natürlich kann aber auch ein eigener Analyzer erstellt werden, sollten die Analyzer von Lucene nicht den gewünschten Anforderungen entsprechen.

Nach dem Laden der Datensätze wird für jede Zeile ein eigens Dokument angelegt. Für die Spalten `article_id`, `headline` und `content` werden eigene Felder erstellt und diese zum Dokument hinzugefügt. Die Felder enthalten den Text der Datenbankobjekte und beinhalten so die eigentlichen Informationen. Außerdem wird schon beim Anlegen des Feldes angegeben, ob das Feld auch im Index gespeichert wird und ob das Feld für die Suche herangezogen wird. Beispielsweise wird das Feld `article_id` zwar im Index gespeichert, jedoch ist es für die Suchergebnisse nicht von Bedeutung.

```
IndexWriter writer = new IndexWriter(INDEX_DIR,
                                     new StandardAnalyzer(),
                                     true);

ResultSet result = pgBaseDao.executeStatement(
                    "SELECT article_id, headline, content
                     FROM article
                     ORDER BY article_id ASC
                     LIMIT 10000 ");

while(result.next()) {
      Document doc = new Document();

      doc.add( new Field("path",
               result.getString("article_id"),
               Field.Store.YES,
               Field.Index.NO));

      doc.add( new Field("headline",
               result.getString("headline"),
               Field.Store.YES,
               Field.Index.TOKENIZED));

      doc.add( new Field("content",
               result.getString("content"),
               Field.Store.YES,
               Field.Index.TOKENIZED));

      writer.addDocument(doc);
}
```

**Listing 6-1: Lucene - Indizierung**

## 6.3 Abfragen

Nach dem Erstellen des Indexes, kann dieser auch schon für Suchen verwendet werden. Dabei ist zu beachten, dass auch hier wieder der `StandardAnalyser`, welcher schon bei der Indizierung verwendet worden ist, zur Anwendung gelangt. Mittels `QueryParser` wird der eingegebene Suchstring in eine Luceneabfrage umgewandelt. Das Durchsuchen des Indexes liefert eine Menge von Treffern, sogenannte Hits. Im folgenden Beispiel wird exemplarisch eine Suche verdeutlicht, welche auf die Überschrift des besten Treffers ausgibt.

```
String index = "C:\\temp\\luceneindex";
String field = "content";
String queryString = "python";

IndexReader reader = IndexReader.open(indexLocation);
Searcher searcher = new IndexSearcher(reader);
Analyzer analyzer = new StandardAnalyzer();

QueryParser parser = new QueryParser(field, analyzer);

Query query = parser.parse(queryString);
Hits hits = searcher.search(query);

Document doc = hits.doc(0);
Field headline = doc.getField("headline");

System.out.println(headline.stringValue());

reader.close();
```

**Listing 6-2: Lucene - Suchabfrage**

Im obigen Beispiel wird nach dem String "`python`" gesucht. Lucene ermöglicht es auch, nach mehreren Wörtern zu suchen. Dabei müssen diese mit logischen Operatoren verknüpft werden. Um alle Texte zu finden, in denen die Wörter „`python`" und „`search`" vorkommen, muss mit dem Suchstring „`python & search`" gesucht werden. Es gibt auch die Möglichkeit einzelne Wörter höher zu gewichten als andere. Weiters kann man mit Wildcards arbeiten. Um den Rahmen dieser Arbeit nicht zu sprengen, verweise ich an dieser Stelle auf die Lucene-Dokumentation.

# 7 Implementierung: PostgreSQL Python Search (PPS)

Der letzte Teil dieser Arbeit widmet sich der Realisierung geeigneter Suchfunktionalitäten in PostgreSQL, die großteils als Pythonfunktionen umgesetzt werden. Daher auch der Name PPS (PostgreSQL Python Search). Wie auch schon bei Tsearch2 und Lucene wird wieder die Artikeltabelle zum Suchen verwendet. Für eine performante Suche wird jeder Artikel in die einzelnen Wörter zerlegt und in eigenen indizierten Tabellen abgespeichert. Bei der Suche wird nicht mehr der Originaltext durchsucht, sondern die neuen Tabellen. Die nachfolgende Grafik gibt einen kurzen Überblick über das Zusammenspiel der einzelnen Teile von PPS.

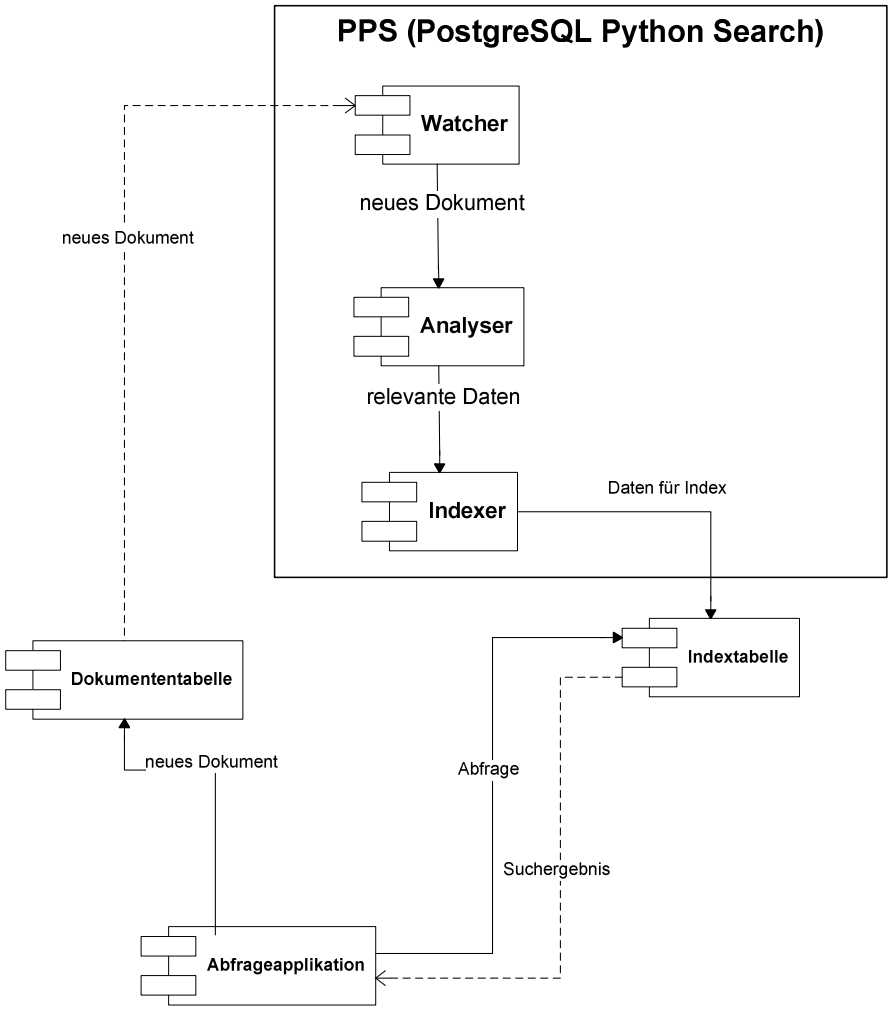


**Abbildung 6: PPS - Ablauf einer Suche und das Zusammenspiel der Komponenten**

Um einen Text in dieser Art und Weise abzuspeichern, gibt es verschiedene Möglichkeiten. In dieser Arbeit werden zwei Versionen vorgestellt. Die erste Variante ist, jedes vorkommende Wort eines Artikels mit der Anzahl, wie oft dieses vorkommt, abzuspeichern. Hier ist dann nur mehr eine Suche nach den Wörtern möglich, ohne dabei Rückschlüsse auf die Positionen innerhalb des Textes zu ziehen. Die zweite Variante, die hier vorgestellt wird, ist jene, dass beim Abspeichern für jedes Wort auch die Position abgespeichert wird. Dies bringt den Vorteil, dass spezielle Ranking-Algorithmen implementiert werden können. Grundlage für diesen Abschnitt der Arbeit stellt das Kapitel „Searching and Ranking“ [Seg07] aus dem Buch „Collective Intelligence“ verfasst von Toby Segaran dar, in dem ein ähnlicher Algorithmus für die Erstellung eines Index, das Durchführen der Suche und das Ranking der Suchergebnisse beschrieben wird. Vor allem für das Ranking der Ergebnisse finden sich dort weitere Informationen.

## 7.1 Version 1

Wie schon erwähnt wird in dieser Implementierungsart festgehalten, wie oft jedes Wort in jedem Artikel vorkommt.

### 7.1.1 Tabellenstruktur

In diesem Abschnitt werden die für die Indizierung der Artikeltabelle notwendigen neuen Tabellen beschrieben. Weiters ist auch eine Änderung an der Artikeltabelle notwendig.

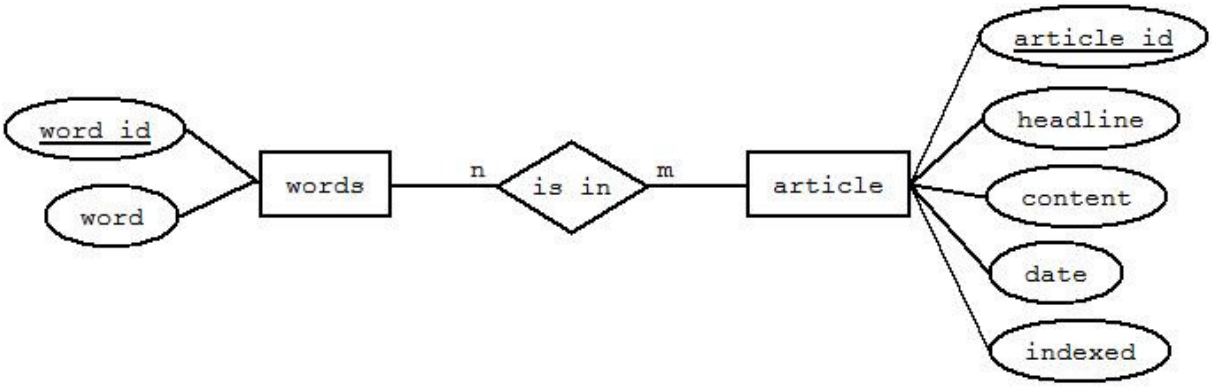


**Abbildung 7: PPS - internes Datenbankschema**

Die Wörter des Ursprungstextes werden in der Tabelle `words` abgespeichert, wobei jedes Wort nur ein einziges Mal vorkommt. Die Spalte word kann somit `unique` deklariert werden. Bei den späteren Suchen wird die Tabelle `words` nach einzelnen Wörtern durchsucht. Deshalb ist es sinnvoll die Spalte `word` zu indizieren.

Die Verbindung zwischen den einzelnen Wörtern und den Artikeln stellt die Entität `word_article` dar. Für jeden Artikel werden hier die vorkommenden Wörter abgespeichert. Als zusätzliche Information wir auch die Häufigkeit, also wie oft ein Wort in einem bestimmten Artikel vorkommt, gespeichert. Für die spätere Implementierung setzen wir noch einen Index auf die Spalte `word_id`.

Zu guter Letzt wird in der Artikeltabelle noch eine Spalte `indexed` hinzugefügt, in der gespeichert wird, wann ein bestimmter Artikel indiziert worden ist. Im folgenden Codelisting sind die Statements zur Erstellung der Tabellen und zur Anlage der entsprechenden Indizes definiert.

```
CREATE TABLE words
(
  word_id serial NOT NULL,
  word character varying(128) NOT NULL,
  CONSTRAINT word_pkey PRIMARY KEY (word_id),
  CONSTRAINT words_word_key UNIQUE (word)
);
```

```
CREATE INDEX idx_word
  ON words
  USING btree
  (word);



CREATE TABLE word_article
(
  article_id integer NOT NULL,
  word_id integer NOT NULL,
  word_count integer NOT NULL,
  CONSTRAINT word_article_pkey PRIMARY KEY(article_id, word_id)
)
WITH (OIDS=FALSE);
ALTER TABLE word_article OWNER TO albert;

CREATE INDEX idx_word_article
ON word_article
USING btree
 (word_id);



ALTER TABLE article ADD COLUMN indexed timestamp;
```

**Listing 7-1: PPS – Anlegen der Tabellen**

### 7.1.2 Indizierung

Sind einmal die Tabellen angelegt, kann auch schon mit der Indizierung begonnen werden. Zu Beginn ist eine Funktion notwendig, welche die Artikeltabelle durcharbeitet und jeden Datensatz zum Index hinzufügt. Das Hinzufügen selbst geschieht innerhalb einer eigenen Methode, die später für die Neuindizierung bei etwaigen Änderungen des Artikels verwendet werden kann. Um die Verarbeitung einfacher zu gestalten, erzeugen wir uns zunächst einen Datentyp der einen Artikel repräsentiert:

```
CREATE TYPE tArticle AS (
     article_id integer ,
     headline character varying(512),
     date date,
     "content" text
);
```

**Listing 7-2: PPS - Datentyp tArticle anlegen**

Die folgende Funktion `indexArticles()` legt den Cursor `cArticle` an. Mit dessen Hilfe wird die Artikeltabelle datensatzweise durchlaufen und jeder Artikel indiziert.

```
CREATE OR REPLACE FUNCTION indexArticles()
RETURNS integer
AS $$

DECLARE
     cArticle NO SCROLL CURSOR FOR
          SELECT * FROM article WHERE indexed IS NULL;
     row tArticle%rowtype;
     rows integer;
BEGIN

PERFORM initIndexing();
rows := 0;
OPEN cArticle;

LOOP
     FETCH NEXT FROM cArticle into row;

     IF NOT FOUND THEN
          EXIT;
     END IF;

     rows:=rows+1;

     PERFORM indexarticle(row);

     UPDATE article SET indexed_simple = now() WHERE CURRENT OF
          cArticle;

END LOOP;

CLOSE cArticle;
PERFORM finishindexing();

RETURN rows;
END;
$$ LANGUAGE plpgsql;
```

**Listing 7-3: PPS - Artikeltabelle indizieren**

Zuerst wird ein Cursor für alle bisher noch nicht indizierten Artikel angelegt. Dabei ist zu beachten, dass der Cursor nur in eine Richtung durchgearbeitet werden kann. Das bringt den Vorteil mit sich, dass die schon gelesenen Datensätze nicht im Speicher gehalten werden und die Verarbeitung somit relativ speicherschonend durchgeführt wird.

Nach dem Öffnen des Cursors werden die noch für die Verarbeitung notwendigen Initialisierungstätigkeiten durchgeführt. Eine genauere Beschreibung folgt bei der Methode `initindexing()`.

Das Auslesen des Cursors wird innerhalb einer Schleife durchgeführt, die so lange läuft, bis kein weiterer Datensatz mehr gelesen werden kann. Für jeden Artikel wird nun die Methode `indexarticle(article tArticle)` aufgerufen und im Anschluss wird noch der aktuelle Timestamp als Indizierungszeit gesetzt. Zu guter Letzt wird noch der Cursor geschlossen und die Methode `finishindexing()` aufgerufen.

Wie schon zuvor beschrieben wurde muss vor jedem Indiziervorgang die Methode `initIndexing()` aufgerufen werden, die für die Verarbeitung eine Liste von Stoppwörtern benötigt. Die Methode liest aus der Tabelle stopwords alle Einträge und speichert jedes Wort in einem Array welches im sogenannten `global dictionary` abgelegt wird. Alle dort gespeicherten Werte sind innerhalb einer Session gültig [Pos08a].

```
CREATE OR REPLACE FUNCTION initIndexing()
RETURNS void
AS $$
#load and store stopwords
 rs = plpy.execute("SELECT stopword FROM stopwords");
 stopwords = []
 for stopword in rs:
  stopwords.append(stopword['stopword'])
 GD["stopwords"] = stopwords
$$ LANGUAGE plpythonu;
```

**Listing 7-4: PPS - initIndexing**

Eine saubere Implementierung verlangt natürlich auch, dass die Liste der Stoppwörter nach der Indizierung wieder freigegeben wird.

```
CREATE OR REPLACE FUNCTION finishIndexing()
RETURNS void
AS $$
#remove stopwords
GD["stopwords"] = None
$$ LANGUAGE plpythonu;
```

**Listing 7-5: PPS - finishIndexing**

Um die Subroutinen der Methode `indexArticles()` zu komplettieren benötigen wir noch die Funktion `indexarticle(article tArticle)`, welche einen Artikel erhält und sich um die Indizierung kümmert. Dabei wird sowohl für den Inhalt als auch für die Überschrift die Methode `indextext_simple(a_id integer, weight integer, words text)` aufgerufen. Dabei ist zu beachten, dass hier für die Überschrift die doppelte Priorität festgelegt wird.

```
CREATE OR REPLACE FUNCTION indexArticle(article tArticle)
RETURNS void
AS $$
BEGIN
 PERFORM indexText(article.article_id, 1, article.content);
 PERFORM indexText(article.article_id, 2, article.headline);
END
$$ LANGUAGE plpgsql;
```

**Listing 7-6: PPS - indexArticle**

Der Hauptteil der Indizierungsfunktionalität ist in der Funktion `indexText(a_id integer, weight integer, words text)` enthalten. Zunächst wird jeder Text mit Hilfe der Funktion `splitWords(words text)` in seine Bestandteile zerlegt. Die einzelnen Wörter werden nun in einer Schleife durchlaufen. Handelt es sich nicht um ein Stoppwort, so wird das Wort in der Tabelle `word_article` eingetragen. Dazu wird zuerst die `Id` des Wortes ausgelesen bzw. vorher automatisch angelegt. Als nächsten Schritt wird festgestellt, ob es schon einen entsprechenden Eintrag in der Tabelle gibt. Ist dies nicht der Fall, so wird eine neue Zeile eingefügt. Diese sagt aus, dass für den aktuellen Artikel das Wort einmal

vorkommt. Hat die vorhergehende Abfrage einen Datensatz gefunden, so wird ein Update auf die Tabelle `word_article` durchgeführt und der `count` um eins erhöht.

```
CREATE OR REPLACE FUNCTION indexText(a_id integer, weight integer,
words text)
RETURNS void
AS $$
DECLARE
 w_id integer;
 word RECORD;
 res word_article%rowtype;
BEGIN

 FOR word in SELECT splitWords(words) AS value LOOP

  CONTINUE WHEN (SELECT isStopWord(word.value));

  w_id = getWordId(word.value);

  SELECT INTO res article_id FROM word_article
          WHERE article_id = a_id AND word_id = w_id;
  IF NOT FOUND THEN
   INSERT INTO word_article (article_id, word_id, word_count)
          VALUES (a_id, w_id, weight);
  ELSE
   UPDATE word_article SET word_count = word_count + weight
          WHERE article_id = a_id AND word_id = w_id;
  END IF;

 END LOOP;

END
$$ LANGUAGE plpgsql;
```

**Listing 7-7: PPS - indexText**

Die Methode `splitWords(words text)` teilt einen Text in seine Bestandteile. Zuerst werden im gesamten Text Sonderzeichen, Zahlen und Satzzeichen am Ende jedes Wortes entfernt. Im Anschluss wird der Text bei den Leerzeichen geteilt und eine Liste der Wörter zurückgegeben.

```
CREATE OR REPLACE FUNCTION splitWords(words text)
RETURNS SETOF text
```

```
AS $$
 import re;
 #lösche Satzzeichen , Zahlen, sonst. Sonderzeichen
 replace = re.compile("[:,;\-\?!'\"\(\)€$%0-9/\\\\]*");
 ret = replace.sub('',words.lower());

 #lösche Satzzeichen am Ende des Textes oder am Ende des Wortes
 replace = re.compile("[\.!?,;:]+($|\W)");
 ret = replace.sub(' ', ret);

 return ret.split();
$$ LANGUAGE plpythonu;
```

**Listing 7-8: PPS - splitWords**

Die Funtion `isStopWord(word varchar)` überprüft für jedes Wort, ob es sich um ein Stoppwort handelt. Ist dies der Fall, so wird `true` zurückgegeben. In dieser Methode wird auch die Länge der Wörter überprüft. Jedes Wort mit weniger als vier Buchstaben wird ebenfalls wie ein Stoppwort behandelt und im Indizierungsvorgang nicht berücksichtigt.

```
CREATE OR REPLACE FUNCTION isStopWord(word varchar)
RETURNS boolean
AS $$
 if word in GD["stopwords"]:
  return True

 if len(word) < 4:
  return True

$$ LANGUAGE plpythonu;
```

**Listing 7-9: PPS - isStopWord**

Die Methode `getWordId (word varchar)` liefert für das als Parameter übergebene Wort die entsprechende `Id` aus der Tabelle `word_id`. Existiert das Wort dort noch nicht, so wird es automatisch eingefügt.

```
CREATE OR REPLACE FUNCTION getWordId (word varchar)
```

```
RETURNS integer
AS $$
 select = "SELECT word_id FROM words WHERE word = $1";
 insert = "INSERT INTO words (word) VALUES ($1)";

 plan_select = plpy.prepare(select, ["varchar"]);
 rv = plpy.execute(plan_select, [word]);

 if len(rv) < 1:
  plan_insert = plpy.prepare(insert, ["varchar"]);
  plpy.execute(plan_insert, [word]);
  rv = plpy.execute(plan_select, [word]);

 return rv[0]['word_id'];

$$ LANGUAGE plpythonu;
```

**Listing 7-10: PPS - getWordId**

### 7.1.3 Änderungen an den Daten

Da unser System auch nach Datenänderungen richtige Ergebnisse liefern muss, werden in diesem Abschnitt zwei Datenbanktrigger erstellen, die auf Veränderungen des Datenbestandes reagieren und unseren Index aktuell halten. Diese Trigger arbeiten jeweils mit einer eigenen Funktion. Dabei ist zu beachten, dass die Methoden zuerst erstellt werden müssen [Pos08a].

Die Funktion `resetIndex()` setzt für einen Artikel zuerst den Indizierungszeitpunkt zurück und löscht im Anschluss alle Einträge des Artikels aus dem Index, also aus der Tabelle `word_article`. Die Verarbeitung des Artikeldatensatzes geschieht über das Objekt `NEW`, welches vom aufrufenden Trigger zur Verfügung gestellt wird. Der Rückgabewert ist vom Typ `trigger` und stellt den geänderten Datensatz dar.

```
CREATE OR REPLACE FUNCTION resetIndex()
RETURNS trigger
AS $$
    BEGIN
     NEW.indexed = NULL;

     DELETE FROM word_article WHERE article_id = NEW.article_id;

     RETURN NEW;
```

```
    END;
$$ LANGUAGE plpgsql;
```

**Listing 7-11: PPS - resetIndex**

Die Methode `resetIndexForDelete()` entspricht im Prinzip der Funktion `resetIndex()`. Der Unterschied besteht darin, dass diese für das Löschen eines Artikels verwendet wird. Aus diesem Grund ist es auch nicht notwendig bzw. auch gar nicht möglich den aktuellen Datensatz zu verändern. Es gibt hier kein Objekt `NEW`, da der Datensatz gelöscht wird. Stattdessen verwenden wir hier das alte Objekt, welches durch die Referenz `OLD` repräsentiert wird. Die einzige Aktivität der Funktion ist das Löschen der Datensätze des zu löschenden Artikels in der Indextabelle `word_article`.

```
CREATE OR REPLACE FUNCTION resetIndexForDelete()
RETURNS trigger
AS $$
    BEGIN
      DELETE FROM word_article WHERE article_id = OLD.article_id;
      RETURN OLD;
    END;
$$ LANGUAGE plpgsql;
```

**Listing 7-12: PPS - resetIndexForDelete**

Die nachfolgende Methode `reindexArticle ()` ist das Gegenstück zur zuvor vorgestellten Funktion `resetIndes()`. Die Verarbeitung des Artikeldatensatzes wird wieder über das Objekt `NEW` durchgeführt. Für die Indizierung des Artikels wird zuerst ein neues Objekt vom Typ `tArticle` angelegt, welches mit den aktuellen Daten des zu ändernden Artikels befüllt werden. Wie auch schon bei der Funktion `indexArticles()` werden vor und nach dem eigentlichen Indizierungsvorgang die Routinen `initindexing()` und `finishindexing()` aufgerufen. Für die Indizierung selbst, kann ebenfalls wieder auf schon vorhandene Methoden zurückgegriffen werden. So wird hier wieder die Funktion `indexArticle(article)` verwendet. Zu guter Letzt wird die Indizierungszeit auf einen aktuellen Timestamp gesetzt. Der Rückgabewert stellt ebenfalls den neuen Datensatz dar.

```
CREATE OR REPLACE FUNCTION reindexArticle()
RETURNS trigger
AS $$
    DECLARE
     article tArticle%rowtype;
    BEGIN
     article.article_id = NEW.article_id;
     article.content = NEW.content;
     article.headline = NEW.headline;
     article.date = NEW.date;

     PERFORM initIndexing();
     PERFORM indexArticle(article);
     PERFORM finishIndexing();

     NEW.indexed = now();
     RETURN NEW;

    END;
$$ LANGUAGE plpgsql;
```

**Listing 7-13: PPS - indexAricle**

Um die drei gerade erstellten Funktionen auch zum richtigen Zeitpunkt aufrufen zu können, benötigen wir für jede dieser Methoden einen geeigneten Trigger. Bei der Namensgebung ist ein kleiner aber wesentlicher Punkt zu beachten. Wenn für ein und dasselbe Event zwei Trigger definiert werden, werden diese in alphabetischer Reihenfolge ausgelöst [Pos08a]. Da wir für eine Datensatzänderung zwei Trigger definieren wollen, benennen wir unsere Trigger einfach mit einem kleinen Präfix und der Methode, die aufgerufen wird. Das Präfix besteht aus einem kleinen `t` und einer aufsteigenden Zahl.

Die Methode `resetIndex()` muss bei jeder Änderung des Datensatzes aufgerufen werden. Bei unseren Triggern geschieht der Funktionsaufruf stets vor der eigentlichen Datenänderung, da auch unsere Datenänderungen  mitübernommen werden sollen.

```
CREATE TRIGGER t1_resetIndex BEFORE UPDATE ON article
            FOR EACH ROW EXECUTE PROCEDURE
            resetIndex();
```

**Listing 7-14: PPS - Trigger: resetIndex**

Wird ein Datensatz gelöscht, so soll die Funktion `resetIndexForDelete()` aufgerufen werden.

```
CREATE TRIGGER t2_resetIndexForDelete BEFORE DELETE ON article
            FOR EACH ROW EXECUTE PROCEDURE
            resetIndexForDelete();
```

**Listing 7-15: PPS - Trigger: resetIndexForDelete**

Für jeden neuen Datensatz soll nun die Methode `reindexArticle()` ausgeführt werden. Dabei spielt es keine Rolle, ob der Artikel durch ein `insert` neu angelegt oder durch ein `update` geändert worden ist. In beiden Fällen ist zu diesem Zeitpunkt kein Eintrag in der Indextabelle vorhanden und wird nun erstellt.

```
CREATE TRIGGER t3_reindexArticle BEFORE UPDATE OR INSERT ON article
            FOR EACH ROW EXECUTE PROCEDURE
            reindexArticle();
```

**Listing 7-16: PPS - Trigger: reindexArticle**

### 7.1.4 Suche

Nachdem im vorhergehenden Kapitel die Indizierung implementiert worden ist, können wir uns nun schon der Suche widmen, die in zwei Schritten durchgeführt wird. Zuerst wird mit Hilfe der Funktion `buildSQL(query varchar)` das SQL-Statement für die Abfrage erzeugt. Im zweiten Schritt wird das SQL-Statement ausgeführt und das Ergebnis zurückgegeben.

Bei der Erzeugung der Abfrage wird der übergebene Querystring wie der Text des Artikels bei der Indizierung behandelt. Der fixe Bestandteil der SQL-Abfrage besteht nur aus der Abfrage der `article_id` aus der Tabelle `article`. Der restliche Teil wird dynamisch aus dem übergebenen Suchstring zusammengebaut. Dazu benötigen wir zunächst eine Liste der

einzelne Wörter, die wieder mit der Funktion `splitWords(words text)` erzeugt wird. In einer Schleife wird diese Liste durchlaufen und für jedes Wort wird überprüft, ob es sich um ein Stoppwort handelt. Ist das der Fall, so wird das Wort nicht in der Abfrage berücksichtigt.

Für jedes Wort werden zwei Joins benötigt. Beim ersten werden aus der Tabelle `article_word` alle Einträge des Artikels hinzugefügt. Mit dem zweiten Join werden aus der Tabelle `words` alle Sätze hinzugefügt, die einen entsprechenden Eintrag in der `word_article` Tabelle besitzen. Natürlich dürfen nicht alle diese Datensätze in das Ergebnis miteinfließen. Die Bedingung wird mit einer LIKE-Abfrage der Spalte `word` erweitert. Es dürfen nur jene Zeilen in das Ergebnis gelangen, die das aktuelle Wort als Teilstring beinhalten. Das Ergebnis enthält neben der eindeutigen Artikel-Id auch den Rang. Dieser wird aus der Summe w`ord_count`-Spalten der einzelnen Wörter berechnet. Da bei der Abfrage mehr als nur eine Wortkombination gefunden werden kann, wird das Ergebnis nach der Spalte `articel_id` gruppiert und die Teilsummen wiederum aufaddiert. Zu guter Letzt werden unsere Treffer noch nach dem errechneten Rank absteigend sortiert.

```
CREATE OR REPLACE FUNCTION buildSQL(query varchar)
RETURNS varchar
AS $$

 fields = '';
 joins = '';
 cond = '';
 wordcount = 0;

 rs = plpy.execute("SELECT splitWords('" + query + "') AS word");
 plpy.execute("SELECT initindexing();");

 for row in rs:
  word = row['word'];

  isstop = plpy.execute("SELECT isStopWord('" + word + "') AS
                        isstopword");
  if isstop[0]['isstopword']:
   continue;

  wordcount += 1;

  if wordcount > 1:
```

```
   cond += " AND ";
   fields += " + ";

  fields += "wa%d.word_count" % (wordcount);

  joins += " INNER JOIN word_article wa%d ON wa%d.article_id =
          a.article_id" % (wordcount, wordcount);
  joins += " INNER JOIN words w%d ON wa%d.word_id = w%d.word_id" %
          (wordcount, wordcount, wordcount);
  word = "%" + word + "%";
  cond += "w%d.word LIKE '%s'" % (wordcount, word);

 plpy.execute("SELECT finishindexing();");

 select = "SELECT a.article_id, sum(%s) AS rank FROM article a %s
           WHERE %s GROUP BY a.article_id ORDER BY rank DESC" %
           (fields, joins, cond);

 return select;

$$ LANGUAGE plpythonu;
```

**Listing 7-17: PPS - SQL erstellen**

Wird die Funktion `buildSQL (query varchar)` mit der Abfrage `'python postgresql'` ausgeführt, wird dabei folgendes SQL erstellt:

```
SELECT a.article_id, sum(wa1.word_count + wa2.word_count) AS rank
FROM article a
INNER JOIN word_article wa1 ON wa1.article_id = a.article_id
INNER JOIN words w1 ON wa1.word_id = w1.word_id
INNER JOIN word_article wa2 ON wa2.article_id = a.article_id
INNER JOIN words w2 ON wa2.word_id = w2.word_id
WHERE w1.word LIKE '%python%' AND w2.word LIKE '%postgresql%'
GROUP BY a.article_id
ORDER BY rank DESC
```

**Listing 7-18: PSS - buildSQL: Beispiel**

Bevor die zweite Funktion implementiert wird, benötigen wir noch einen Datentyp, welcher einen Treffer repräsentiert. Dieser Typ besteht aus der Artikel-Id und dem Rang, der die Wichtigkeit des Treffers darstellt.

```
CREATE TYPE tHit AS (
     article_id integer ,
     rank float
);
```

**Listing 7-19: PPS - Datentyp tHit anlegen**

Mit der Funktion query wird nun die Suche durchgeführt. Nach dem Anlegen der SQL-Abfrage wird dieses abgesetzt und das Ergebnis aufbereitet zurückgegeben. Um die Treffer besser vergleichbar zu machen werden wir den Rang noch normalisieren. Da die einzelnen Treffer schon sortiert in der Ergebnisliste sind, befindet sich in der ersten Zeile der beste Eintrag, also jener bei dem die einzelnen Wörter der Suchabfrage am Häufigsten vorkommen. Den Rang der einzelnen Treffer werden wir nun einfach durch den Rang des ersten dividieren. Der Artikel mit der höchsten Relevanz für uns hat nun also den Rang eins und die weiteren Artikel je nachdem wie oft die einzelnen Suchwörter gefunden wurden einen Wert zwischen null und eins.

```
CREATE OR REPLACE FUNCTION query(query varchar)
RETURNS SETOF tHit
AS $$

 #fetch data
 select = plpy.execute("SELECT buildSQL_simple('" + query + "') AS
                        sql");
 rs = plpy.execute(select[0]["sql"]);

 hits = [];

 if (len(rs) > 0):
  max = 1.0 * rs[0]["rank"];

 for row in rs:
  hit = {};
  hit["article_id"] = row["article_id"];
  hit["rank"] = row["rank"]/max;
  hits.append(hit);

 return hits;
$$ LANGUAGE plpythonu;
```

**Listing 7-20: PPS - Abfrage**

Da ein Ergebnis mit dem Datentyp `tHit` möglicherweise umständlich in der Verwendung ist, werden nun zwei weitere Funktionen zur Extraktion der eigentlichen Daten vorgestellt. Es wird nun eine Methode `getArticle_id (hit tHit)`, die von einem Objekt des Types `tHit` die Artikel-Id liefert, und eine Methode `getRank (hit tHit)`, welche den Rang eines Artikels extrahiert, implementiert. Im nachfolgenden Codeblock befindet sich auch ein Beispiel zur Anwendung der beiden Funktionen. Dabei wird diese für jeden Datensatz aufgerufen.

```
CREATE OR REPLACE FUNCTION getArticle_id (hit tHit)
RETURNS integer
AS $$
 return hit["article_id"];
$$ LANGUAGE plpythonu;

CREATE OR REPLACE FUNCTION getRank (hit tHit)
RETURNS float
AS $$
 return hit["rank"];
$$ LANGUAGE plpythonu;

SELECT getArticle_id(hit) AS article_id, getRank(hit) AS rank FROM
query('trade europe london') hit
```

**Listing 7-21: PPS - Datenaufbereitung**

## 7.2 Version 2

Die soeben implementierte Suche erlaubt es in den Texten nach verschiedenen Wörtern zu suchen. Allerdings ist es nicht möglich, genaue Rückschlüsse auf die Relevanz der einzelnen Treffer zu ziehen. Darum wird hier eine zweite Version der Suche vorgestellt, die für jedes Wort auch die Position innerhalb des Textes mitspeichert. Somit ist es möglich, dass verschiedene Faktoren, wie die Position innerhalb der Texte oder den Abstand der einzelnen Suchwörter zueinander, in den Ranking-Algorithmus miteinfließen zu lassen.

Viele der schon in Version 1 implementierten Funktionen können hier ohne Änderungen wiederverwendet werden. Im folgenden Kapitel werden nur jene Methoden beschrieben, die für die Adaptierung notwendig sind.

### 7.2.1 Tabellenstruktur

Um die genauen Positionen mit abzuspeichern, werden nun zwei Tabellen mit der gleichen Struktur benötigt. Die neue Tabellenstruktur sieht folgendermaßen aus:

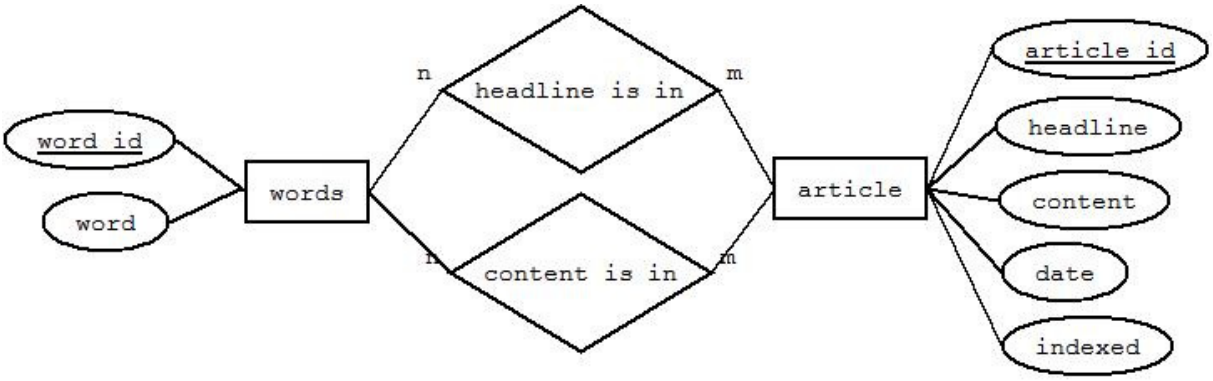


**Abbildung 8: PPSv2- internes Datenbankschema**

Im nachfolgenden Codelisting werden die für die neuen Tabellen benötigten Create-Statements angeführt. Des Weiteren werden noch zwei Indizes auf den neuen Tabellen benötigt. Für eine performante Suche wird jeweils die Spalte `word_id` der Tabellen `word_position_headline` und `word_position_content` indiziert.

```
DROP TABLE IF EXISTS word_position_content;

CREATE TABLE word_position_content
(
 article_id integer NOT NULL,
 word_id integer NOT NULL,
 "position" integer NOT NULL,
 CONSTRAINT word_position_content_pkey PRIMARY KEY(article_id,
word_id, "position")
)
WITH (OIDS=FALSE);
```

```
ALTER TABLE word_position_content_pkey OWNER TO albert;

CREATE INDEX idx_word_position_content
 ON word_position_content
 USING btree
 (word_id);


DROP TABLE IF EXISTS word_position_headline;

CREATE TABLE word_position_headline
(
 article_id integer NOT NULL,
 word_id integer NOT NULL,
 "position" integer NOT NULL,
 CONSTRAINT word_position_headline_pkey PRIMARY KEY(article_id,
word_id, "position")
)
WITH (OIDS=FALSE);
ALTER TABLE word_position_headline OWNER TO albert;

CREATE INDEX idx_word_position_headline
 ON word_position_headline
 USING btree
 (word_id);
```

**Listing 7-22: PPSv2 - word_position_content**

### 7.2.2 Indizierung

Die meisten Funktionen können unverändert weiterverwendet werden. Die einzigen Änderungen für die Adaptierung konzentrieren sich auch die Funktion `indextext(a_id integer, tablename varchar, words text)`. Jetzt werden wir nicht nur für jedes unterschiedliche Wort, sondern für jedes Wort einen eigenen Datensatz erstellen. Dafür wird nur eine Schleife benötigt, die die übergebene Wortliste durchläuft. Falls es sich nicht um ein Stoppwort handelt, wird für jedes Wort ein Eintrag in der Indextabelle, deren Bezeichnung sich aus dem String `word_position` und dem Funktionsparameter `table` zusammensetzt, hinzugefügt.

```
CREATE OR REPLACE FUNCTION indextext(a_id integer, tablename
varchar, words text)
```

```
RETURNS integer
AS $$
DECLARE
 w_id integer;
 word RECORD;
 pos integer;
BEGIN
 pos := 0;
 FOR word in SELECT splitwords(words) AS value LOOP
  pos := pos + 1;
  CONTINUE WHEN (SELECT isstopword(word.value));

  w_id = getwordid(word.value);
  EXECUTE 'INSERT INTO word_position_' || tablename || '(article_id,
           word_id, position) VALUES (' || a_id || ', ' || w_id ||
           ', ' || pos || ');' ;

 END LOOP;
 return pos;
END
$$ LANGUAGE plpgsql;
```

**Listing 7-23: PPSv2 - indextext**

### 7.2.3 Änderungen an den Daten

Um den Datenbestand aktuell zu halten, können bis auf zwei Anpassungen der Triggerfunkionen der der Version 1 verwendet werden. Die Funktionen `resetIndex()` und `resetIndexForDelete()` müssen dahingehend geändert werden, dass die Einträge aus den Tabellen `word_position_content` und `word_position_headine` anstatt der Tabelle `word_article` gelöscht werden.

```
CREATE OR REPLACE FUNCTION resetIndex()
RETURNS trigger
AS $$
    BEGIN
      NEW.indexed = NULL;

      DELETE FROM word_position_headline WHERE article_id =
           NEW.article_id;
      DELETE FROM word_position_content WHERE article_id =
           NEW.article_id;

      RETURN NEW;
```

```
    END;
$$ LANGUAGE plpgsql;
```

**Listing 7-24: PPSv2 – resetIndex**

```
CREATE OR REPLACE FUNCTION resetIndexForDelete()
RETURNS trigger
AS $$
    BEGIN
      DELETE FROM word_position_headline WHERE article_id =
           OLD.article_id;
      DELETE FROM word_position_content WHERE article_id =
           OLD.article_id;
      RETURN OLD;
    END;
$$ LANGUAGE plpgsql;
```

**Listing 7-25: PPS - resetIndexForDelete**

### 7.2.4 Suche

Auch in dieser Version wird die Suche in die zwei Schritte Erzeugung des SQL's und die Durchführung inklusive der Datenaufbereitung gegliedert. Unter der Datenaufbereitung wird hier vor allem die Sortierung nach der Relevanz verstanden, wobei in dieser Version weit komplexere Rankingalgorithmen einfließen.

Die wesentliche Änderung der Funktion `buildSQL(query varchar)` ist, dass hier ein weiterer Join benötigt wird. Wie schon bei der vorhergehenden Version werden für jedes Wort Joins zum SQL-Statement hinzugefügt. Zuerst werden aus der Tabelle `word_position_content` all jene Zeilen benötigt, die dem jeweiligen Wort entsprechen.

Da auch die Überschriften der Artikel bei der Suche berücksichtigt werden müssen, werden auch die Daten der Tabelle `word_position_headline` einbezogen. Dies geschieht genauso wie bei der Tabelle `word_position_content`. Das Ergebnis wird mittels `UNION ALL` zusammengefügt.

```
CREATE OR REPLACE FUNCTION buildSQL (query varchar)
```

```
RETURNS varchar
AS $$

 fields = '';
 joins = '';
 cond = '';
 wordcount = 0;

 rs = plpy.execute("SELECT splitWords('" + query + "') AS word");
 plpy.execute("SELECT initindexing();");

 for row in rs:
  word = row['word'];
  wordcount += 1;
  fields += ", wp%d.position AS pos%d" % (wordcount, wordcount);
  joins += " LEFT JOIN word_position_content wp%d ON a.article_id =
                wp%d.article_Id" % (wordcount, wordcount);
  joins += " INNER JOIN words w%d ON wp%d.word_id = w%d.word_id" %
                (wordcount, wordcount, wordcount);

  if wordcount > 1:
   cond += " AND ";
  word = "%" + word + "%";
  cond += "w%d.word LIKE '%s'" % (wordcount, word);

 plpy.execute("SELECT finishindexing();");

 select = "SELECT a.article_id, %d AS type %s FROM article a %s
           WHERE %s " % (1, fields, joins, cond);
 joins = joins.replace('content', 'headline');
 select += "UNION ALL SELECT a.article_id, %d AS type %s FROM
            article a %s WHERE %s" % (2, fields, joins, cond);

 return select;

$$ LANGUAGE plpythonu;
```

**Listing 7-26: PPSv2 - SQL erstellen**

Wird die Funktion `buildSQL (query varchar)` mit der Abfrage `'python postgresql'` ausgeführt, wird dabei folgendes SQL erstellt:

```
SELECT a.article_id, 1 AS type , wp1.position AS pos1
FROM article a
LEFT JOIN word_position_content wp1 ON a.article_id = wp1.article_Id
INNER JOIN words w1 ON wp1.word_id = w1.word_id
```

```
WHERE w1.word LIKE '%python%'
UNION ALL SELECT a.article_id, 2 AS type , wp1.position AS pos1
FROM article a
LEFT JOIN word_position_headline wp1 ON a.article_id =
wp1.article_Id
INNER JOIN words w1 ON wp1.word_id = w1.word_id
WHERE w1.word LIKE '%python%'
```

**Listing 7-27: PSSv2 - buildSQL: Beispiel**

Mit der Funktion `query` wird auch hier wieder die Suche durchgeführt. Nach dem Anlegen der SQL-Abfrage wird diese abgesetzt und das Ergebnis aufbereitet zurückgegeben. Bei dem verwendeten Rankingalgorithmus werden drei Faktoren, die auf dem Inhalt der einzelnen Artikel beruhen, berücksichtigt [Seg07]:

1. **Anzahl**

Mit einer großen Wahrscheinlichkeit sind jene Texte für den Anwender von hoher Bedeutung, in denen die Suchwörter öfters vorkommen. In der Funktion `query (query varchar)` hat auch dieser Faktor den meisten Einfluss auf den Rang, also auf die Anzahl der gefundenen Treffer, nämlich 60%.

2. **Abstand**

Wenn eine Suche aus mehreren Wörtern besteht, sind Resultate bei denen sich die Abfragewörter nahe beisammen befinden oftmals für den Anwender von höherer Bedeutung. Dieser Faktor fließt mit 20% in die Berechung des Endranges ein.

3. **Position**

Das Hauptthema eines Artikels befindet sich meist schon am Beginn des Textes. Der dritte Einflussfaktor der mit 20% in den Rang miteinfließt, ist also die Position innerhalb des Textes.

Bei der Berechnung des Ranges wird zunächst für alle drei Faktoren der beste Wert für jeden Text und auch der beste Wert aller Texte berechnet. Um diese Werte miteinander vergleichen

zu können, werden diese im nächsten Schritt normalisiert. Beim Faktor Anzahl wird der Wert jedes Artikels durch den besten Wert, also die höchste Anzahl dividiert. Bei den Faktoren Abstand und Position wird der Wert jedes Textes durch den Besten, also den absolut kleinsten Wert dividiert.

Die so berechneten Werte werden im Anschluss noch gewichtet und aufaddiert. Zu guter Letzt muss noch nach dem Rang absteigend sortiert werden.

```
CREATE OR REPLACE FUNCTION query (query varchar)
RETURNS SETOF tHit
AS $$

 from operator import itemgetter

 #fetch data
 select = plpy.execute("SELECT buildSQL('" + query + "')");
 rs = plpy.execute(select[0]["buildsql"]);
 scores = {};

 #set min and max values
 min_val = 0.000001;
 max_val = 999999.9;

 min_pos_sum = max_val;
 min_abstand = max_val;
 max_anzahl = 0;

 word_count = 0;
 if (len(rs) > 0):
  word_count = len(rs[0]) - 2;

 pos_x = ['pos%d' % x for x in xrange(1, word_count + 1)];

 type = 1;

 for row in rs:
   type = row["type"];
   if row["article_id"] in scores:
     scores[row["article_id"]][0] += 1 * type;
   else:
     scores[row["article_id"]] = [1.0 * type, max_val, max_val];

   posvals = [];
   pos_sum = 0.0;
```

```
   for pos in pos_x:
    posvals.append(row[pos]);
    pos_sum += row[pos];

   posvals.sort();
   abstand = float(posvals[word_count-1] - posvals[0]) / type;
   pos_sum /= type;

   if scores[row["article_id"]][1] > pos_sum:
    scores[row["article_id"]][1] = pos_sum;

   if scores[row["article_id"]][2] > abstand:
    scores[row["article_id"]][2] = abstand;


   if (min_abstand > abstand):
    min_abstand = abstand;

   if (min_pos_sum > pos_sum):
    min_pos_sum = pos_sum;

   if (max_anzahl < scores[row["article_id"]][0]):
    max_anzahl = scores[row["article_id"]][0];

 hits = [];
 for article_id in scores:
 #for article_id, vals in scores:
  (anzahl, pos_sum, abstand) = scores[article_id];

  #calculate values vor ranking
  v1 = anzahl / max_anzahl;
  v2 = min_pos_sum / max(min_val, pos_sum);
  v3 = min_abstand / max(min_val, abstand);


  #build result-types
  hit = {};
  hit["article_id"] = article_id;
  hit["rank"] = 0.6*v1 + 0.2*v2 + 0.2*v3;
  hits.append(hit);


 #sort results
 hits = sorted(hits, key=itemgetter("rank"), reverse=True);

  return hits;
$$ LANGUAGE plpythonu;
```

**Listing 7-28: PPSv2 – Abfrage**

# 8 Test

In diesem Kapitel werden die erzielten Ergebnisse mit dem Tsearch2-Framework mit den Resultaten von PPS verglichen.

## 8.1 Testplanung

Um die Ergebnisse zu testen, werden nun einige Testfälle definiert. Der erste Teil des Tests ist eine Funktionsüberprüfung mit generierten Texten. Der zweite Teil durchsucht die Daten der Artikeltabelle.

### 8.1.1 Funktionstest

Für den Test wird nicht die gesamte Artikeltabelle indiziert, sondern es werden nur 10.000 Einträge verwendet. Zusätzlich werden weitere Artikel angelegt, nach denen gesucht wird. Der Inhalt dieser Artikel wird so gewählt, dass die vorkommenden Wörter nicht in einem Wörterbuch aufzufinden sind, also eine zufällige Buchstabenkombination. Beim Anlegen der Artikel wird überprüft, ob die Artikel automatisch indiziert werden. Da im nachfolgenden Test nur SQL-Abfragen durchgeführt werden, kann auch auf einfache Software, nämlich pgAdminIII[1] zurückgegriffen werden. Als Testsystem wird ein Fujitsu Computer Siemens Notebook[2] verwendet.

Bei dem Test werden folgende Schritte durchgeführt:

1. Indizierung der ersten 10.000 Artikel
    - Bei der Indizierung muss darauf geachtet werden, dass die Datenbanktrigger, die die Daten aktuell halten, für diesen Schritt vorübergehend gelöscht werden, da sonst jeder Datensatz doppelt bearbeitet wird.

[1] pgAdminIII, Version 1.8.3 (4. Juni 2008)
[2] Hardware: Intel Dual Core 1,73GHz; Arbeitsspeicher 1GB Arbeitsspeicher

2. Anlegen von drei Artikeln

    - Mittels Java-Programm wird der Inhalt der Artikel generiert. Sowohl der Text als auch der Inhalt des Artikels wird dabei automatisch erstellt. Danach wird werden diese Daten jeweils 1.000-mal in die Datenbank eingefügt und die automatische Indizierung wird überprüft.

3. Suche durchführen

    - Die Suche soll mit einem eindeutigen Wort aus dem ersten bei Punkt 2 erstellten Artikel durchgeführt werden. Im Ergebnis darf daher auch nur dieser eine Artikel enthalten sein (1.000-mal).

4. Artikel ändern

    - Im soeben verwendeten Artikel soll das Suchwort verändert werden, so dass dieser bei einer neuerlichen Suche nicht mehr gefunden wird.

5. Suche durchführen

    - Durchführen einer Suche mit identischem Querystring wie bei Punkt 3. Als Ergebnis wird nun eine leere Menge erwartet.

    - Durchführen einer neuen Suche mit dem unter Punkt 4 verändertem Wort. Das Resultat muss nun wieder 1.000-mal der erste Artikel sein.

6. Artikel löschen

    - Nun soll der gesamte Artikel gelöscht werden, wobei die Daten automatisch aus dem Index entfernt werden müssen.

7. Suche durchführen

    - Eine neuerliche Suche wie unter Punkt 5 darf zu keinem Ergebnis mehr führen.

### 8.1.2 Performancetest

Dieser Teil des Tests verwendet nur ausschließlich vorhandene Texte. Zuerst werden eine 50.000 Artikel indiziert und im Anschluss wird nach diesen gesucht. Dabei muss wieder

berücksichtig werden, dass für die manuelle Indizierung die Update-Trigger vorübergehend gelöscht werden. Gesucht wird nach folgenden Wörtern:

- confounding expectations mexico capel

  Mit dieser Suchkombination soll nur ein bestimmter Text gefunden werden, in dem diese Wörter vorkommen. In den anderen Texten kommt diese Kombination nicht vor.

- confounding expectations mexico

  Auch mit dieser Kombination wird wieder nur derselbe Text gefunden.

- confounding expectations

  Mit dieser Kombination werden schon mehrere Texte gefunden, jedoch noch immer eine geringe Anzahl.

- confounding
- money

## 8.2 Testdurchführung

In diesem Abschnitt werden die im vorherigen Kapitel geplanten Testfälle durchgeführt, wobei die Testfälle gegen die Implementierungen Tsearch2 und die zwei PPS-Algorithmen getestet werden.

### 8.2.1 Funktionstest

Mit der nachfolgenden Funktion `insertTestArticles()` werden 10.000 Artikel in die Tabelle eingefügt. Die Inhalte sind zufällig generierte Texte.

```
CREATE OR REPLACE FUNCTION insertTestArticles() RETURNS VARCHAR AS
$$
DECLARE
     anzahl integer;
     start integer;
```

```
      sql varchar(1100);

BEGIN
      start := 900000;
      anzahl := 0;
      WHILE anzahl < 3000 LOOP
            anzahl := anzahl + 1;
            INSERT INTO article (article_id, date, headline, content)
                  VALUES (start + anzahl, '2008-12-15', ' ahqkgooei
                  ojthrlqvl …', '…');

            anzahl := anzahl + 1;
            INSERT INTO article (article_id, date, headline, content)
                  VALUES (start + anzahl, '2008-12-15', ' whufcixgk
                  ccwtzcdzi …', '…');
            anzahl := anzahl + 1;
            INSERT INTO article (article_id, date, headline, content)
                  VALUES (start + anzahl, '2008-12-15', ' vmkmpnafj
                  qomnyttsd …', '…');
      END LOOP;
      RETURN anzahl;
END;
$$ LANGUAGE plpgsql;
```

**Listing 8-1: Funktionstest - insertTestArticles**

Bei der Durchführung des Funktionstests gab es keinerlei Abweichungen von den erwarteten Ergebnissen. Die genauen SQL-Statements der Testfälle befinden sich im Anhang (Listing 11-1: Funktionstest - Tsearch2, Listing 11-2: Funktionstest – , Listing 11-3: Funktionstest - PPSv2). In der nachfolgenden Tabelle sind die Zeiten der Abfragen ersichtlich:

| | **Dauer der Durchführung [in ms]** | | |
|---|---|---|---|
| | **Tsearch2** | **PPS** | **PPSv2** |
| Indizierung der ersten 10.000 Artikel | 1.918.535 | 1.535.936 | 1.289.151 |
| Anlegen und Indizierung von 3.000 Artikel | 820.403 | 2.150.102 | 2.850.497 |
| Suche nach der ersten Artikelgruppe (1.000 Resultate) | 15.553 | 3.417 | 374 |
| Artikel ändern | 71.557 | 535.850 | 528.613 |
| Ursprüngliche Suche (0 Resultate) | 47 | 842 | 128 |
| Adaptierte Suche (1.000 Resultate) | 15.116 | 203 | 154 |

| | | | |
|---|---|---|---|
| Artikel löschen | 12.224 | 99.481 | 97.541 |
| Artikel suchen (0 Resultate) | 162 | 125 | 224 |

**Abbildung 9: Funktionstest – Zusammenfassung**

Aus dieser Tabelle ersichtlich ist, dass Tsearch2 beim Indizieren der Echtdaten etwas langsamer ist als die zwei PPS-Implementierungen, allerdings bei der Indizierung bzw. Änderung der generierten Daten doch spürbar schneller ist. Bei der Durchführung der Suchen zeigen sich signifikante Unterschiede in der Ausführungszeit. Hier sind die beiden PPS-Algorithmen merklich schneller als die Tsearch2-Variante.

### 8.2.2 Performancetest

Nach dem Funktionstest wird nun auch der Perfomancetest durchgeführt. Dabei werden die ersten 50.000 noch nicht indizierten Artikel in den Index aufgenommen. Die genauen Testergebnisse finden sich im Anhang wieder (Listing 11-4: Performancetest - Tsearch2, Listing 11-5: Performancetest – PPS, Listing 11-6: Performancetest – PPSv2). Die folgenden Tabellen fassen die im Test gemessenen Zeiten zusammen:

| **Indizierung** | **Indizierungszeit [in ms]** | | |
|---|---|---|---|
| | **Tsearch2** | **PPS** | **PPSv2** |
| 50.000 Artikel | 14.263.784 | 8.237.567 | 7.245.789 |

| **Suche nach** | **Anzahl der Treffer** | **Suchzeit [in ms]** | | |
|---|---|---|---|---|
| | | **Tsearch2** | **PPS** | **PPSv2** |
| confounding expectations mexico capel | 1 | 131 | 452 | 3.993 |
| confounding expectations mexico | 1 | 215 | 172 | 265 |
| confounding expectations | 15 | 294 | 156 | 281 |
| confounding | 27 | 156 | 218 | 265 |
| money | 4017 | 45.911 | 49.124 | 52.696 |
| exchange | 6926 | 15.603 | 39.765 | 10.142 |

**Abbildung 10: Performancetest - Zusammenfassung**

Wie in dieser Tabelle ersichtlich, liegen die erhaltenen Antwortzeiten ziemlich nahe beisammen. Es zeigt sich, dass bei einer hohen Trefferanzahl die Antwortzeit rapide zunimmt.

Weiters auffällig sind auch die Suchzeiten mit PPSv2. Mit dem Suchstring „`confounding expectations mexico capel`“ wird eine relativ hohe Antwortzeit erreicht, wobei mit „`exchange`“ gegenüber den anderen Implementierungen eine geringere Durchführungszeit erreicht wird. Die Unterschiede resultieren in der unterschiedlichen Laufzeit der für die Suche notwendigen SQL-Abfragen, wobei beim Wiederholen, trotz aktualisieren des Queryplans mit `ANALYZE VACUUM`, der Abfragen starke Schwankungen auftreten.

# 9 Zusammenfassung und Ausblick

In dieser Arbeit wurde zunächst eine theoretische Grundlage und im Anschluss einige mögliche Implementierungen zur Volltextsuche vorgestellt. Der Aufwand für eine eigenständige Implementierung der Volltextsuche in der Datenbank ist natürlich wesentlich höher als wenn man auf schon existierende Lösungen zurückzugreifen.

Ohne hohen Aufwand wird eine selbst entwickelte Volltextsuche nicht an die des Frameworks Lucene herankommen. Beim Einsatz von Lucene ergeben sich aber einige nicht zu vernachlässigende Nachteile. Da der Index nicht in der Datenbank gespeichert ist, können auch Updates nur erschwert durchgeführt werden. So müssen alle Applikationen die den Datenbestand verändern auch immer den Lucene-Index aktualisieren. Hier ergibt sich bei Technologieänderungen, wegen der Verflechtung der Programmlogik mit der Datenbank, ein erheblicher Änderungsaufwand. Alternativ könnte auch bei jeder Datenänderung ein Trigger ausgelöst werden, der mittels Messagingsystem, Webservice oder ähnlichen Technologien das Update des Lucene-Index veranlasst. Ist auch das nicht möglich, so kann der Index noch immer periodisch upgedatet werden, wobei aber in Kauf genommen werden muss, dass sich der Index zumindest vorübergehend nicht auf dem aktuellen Stand befindet.

Erweiterungsmöglichkeiten von PPS gibt es vor allem im des Bereich Ranking. In den in dieser Arbeit vorgestellten Methoden fließen keine Anwenderinformationen mit ein. So

könnten etwa Bewertungen von Anwendern oder Verlinkungen der Texte in die Berechnung des Ranges miteinfließen.

# 10 Literaturverzeichnis


[BER99] GiST: A Generalized Search Tree for Secondary Storage, August 1999. http://gist.cs.berkeley.edu/gist1.html, zuletzt besucht am 12.02.2009

[IRS00] Jürgen Heber. Knowledge Discovery, Kapitel 3 – Information Retrieval Systeme, Juni 2000. http://www.iicm.tugraz.at/Teaching/theses/2000/_idb96_/jheber/html/kapitel3.html, zuletzt besucht am 08.02.2009

[Luc08a] Apache Foundation, Lucene Java Documentation, November 2008. http://lucene.apache.org/java/2_4_0/, zuletzt besucht am 13.02.2009

[Luc08b] Apache Foundation, Lucene-java Wiki, Oktober 2008. http://wiki.apache.org/lucene-java, zuletzt besucht am 18.02.2009

[Neo08] Oliver Thewalt. Neogrid, 2008. http://neogrid.de/EDV-Lexikon.php?Feld=Lexikalische-Grundeinheit+Lexikalische-Grundeinheit+Lexikalische-Grundeinheit&PHPSESSID=s4cq37s260cscoaa44scicp8p0, zuletzt besucht am 07.10.2008

[Pos08a] PostgreSQL Global Development Group. PostgreSQL 8.3.3 Documentation, 2008. http://www.postgresql.org/docs/8.3/static/, zuletzt besucht am 13.10.2008.

[Pos08b] Cornelia Boenigk und Ralf Burger. Postgres – Deutsche Projekt Seite, 2009. http://www.postgres.de/, zuletzt besucht am 08.02.2009

[Ris09] C.J. van Risjsbergen, Information Retrieval, Kapitel Evaluation, 1979. http://www.dcs.gla.ac.uk/Keith/Chapter.7/Ch.7.html, zuletzt besucht am 02.03.2009

[Sch06] Georg Schumann. Lucene, Volltextsuche mit Java, 5. Juli 2006. http://cms.hs-augsburg.de/report/2006/Schumann_Georg__Lucene/Lucene.pdf, zuletzt besucht am 07.10.2008

[Sol08] Ray Hon, Powerful Full Text Search Engine - Part 1 Lucene Introduction, 4. Juli 2008. http://www.solutionhacker.com/2008/07/04/powerful-full-text-search-engine-lucene-part-1/, zuletzt besucht am 08.02.2009

[Seg07] Toby Segaran. Collective Intelligence - Building Smart Web 2.0 Applications, Kapitel Searching and Ranking, Seiten 54-85, O'Reilly, 2007

[Sig06] Teodor Sigaev, Oleg Bartunov. Gin for PostgreSQL, April 2006. http://www.sigaev.ru/gin/README.txt, zuletzt besucht am 12.02.2009

[Bar08] Oleg Bartunov. The Tsearch2 Guide, Oktober 2006. http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/docs/tsearch2-guide.html, zuletzt besucht am 13.02.2009

[She01] Chris Sherman. Google Goes for Stopwords, Dezember 2001. http://searchenginewatch.com/2158311, zuletzt besucht am 03.03.2009

[lip06] The Linux Information Project. Scalable Definition, März 2006. http://www.linfo.org/scalable.html, zuletzt besucht am 03.03.2009

[wis09] Der invertierte Index und das erweiterte boolesche Modell. http://wissensexploration.de/wissensexploration-web-ir-invertierterindex.php, zuletzt besucht am 12.02.2009

# 11 Anhang

## 11.1 Abbildungsverzeichnis

## 11.2 Codelistingverzeichnis

## 11.3 Testdurchführung

```
UPDATE article
SET content_tsvector =
     to_tsvector('english', content)
      || setweight(to_tsvector('english', headline), 'C')
 WHERE article_id >= 2286
 AND article_id <=12642;
--Abfrage war erfolgreich durchgeführt: 10000 Zeilen, 1918535 ms
Ausführungszeit.

SELECT insertTestArticles();
--Gesamtlaufzeit der Abfrage: 820403 ms: (beim Einfügen von 3.000
Artikeln a 10.000 Zeichen)

SELECT * FROM findArticle('ojthrlqvl');
--Gesamt-laufzeit der Abfrage: 15553 ms. 1000 Zeilen geholt.

UPDATE article SET headline = ' ahqkgooei ojthtestrlqvl …'
WHERE article_id >= 900000 AND headline = ' ahqkgooei ojthrlqvl …'
--Abfrage war erfolgreich durchgeführt: 1000 Zeilen, 71557 ms
Ausführungszeit.

SELECT * FROM findArticle('ojthrlqvl');
--Gesamtlaufzeit der Abfrage: 47 ms. 0 Zeilen geholt.

SELECT * FROM findArticle('ojthtestrlqvl');
--Gesamtlaufzeit der Abfrage: 15116 ms. 1000 Zeilen geholt.

DELETE FROM article WHERE article_id >= 900000
--Abfrage war erfolgreich durchgeführt: 3000 Zeilen, 12224 ms
Ausführungszeit.

SELECT * FROM findArticle('ojthtestrlqvl');
--Gesamtlaufzeit der Abfrage: 162 ms. 0 Zeilen geholt.
```

```
SELECT * FROM findArticle('ojthrlqvl');
--Gesamtlaufzeit der Abfrage: 131 ms. 0 Zeilen geholt.
```

**Listing 11-1: Funktionstest - Tsearch2**

```
SELECT cArticle();
--Gesamtlaufzeit der Abfrage: 1535936 ms. 1 Zeilen geholt.

SELECT insertTestArticles();
--Gesamtlaufzeit der Abfrage: 2150102 ms.

SELECT * FROM query_simple('ojthrlqvl');
--Gesamtlaufzeit der Abfrage: 3417 ms. 1000 Zeilen geholt.

UPDATE article SET headline = ' ahqkgooei ojthtestrlqvl …'
WHERE article_id >= 900000 AND headline = ' ahqkgooei ojthrlqvl …'
--Abfrage war erfolgreich durchgeführt: 1000 Zeilen, 535850 ms
Ausführungszeit.

SELECT * FROM query_simple('ojthrlqvl');
--Gesamtlaufzeit der Abfrage: 842 ms. 0 Zeilen geholt.

SELECT * FROM query_simple('ojthtestrlqvl');
--Gesamtlaufzeit der Abfrage: 203 ms. 1000 Zeilen geholt.

DELETE FROM article WHERE article_id >= 900000
--Abfrage war erfolgreich durchgeführt: 3000 Zeilen, 99481 ms
Ausführungszeit.

SELECT * FROM query_simple('ojthrlqvl');
--Gesamtlaufzeit der Abfrage: 125 ms. 0 Zeilen geholt.

SELECT * FROM query_simple('ojthtestrlqvl');
--Gesamtlaufzeit der Abfrage: 274 ms. 0 Zeilen geholt.
```

**Listing 11-2: Funktionstest – PPS**

```
SELECT cArticle();
--Gesamtlaufzeit der Abfrage: 1289151 ms. 1 Zeilen geholt.

SELECT insertTestArticles();
--Gesamtlaufzeit der Abfrage: 2850497  ms.

SELECT * FROM query('ojthrlqvl');
--Gesamtlaufzeit der Abfrage: 374 ms. 1000 Zeilen geholt.

UPDATE article SET headline = ' ahqkgooei ojthtestrlqvl …'
```

```
WHERE article_id >= 900000 AND headline = ' ahqkgooei ojthrlqvl …
--Abfrage war erfolgreich durchgeführt: 1000 Zeilen, 528613 ms
Ausführungszeit.

SELECT * FROM query('ojthrlqvl');
--Gesamtlaufzeit der Abfrage: 128 ms. 0 Zeilen geholt.

SELECT * FROM query('ojthtestrlqvl');
--Gesamtlaufzeit der Abfrage: 154 ms. 1000 Zeilen geholt.

DELETE FROM article WHERE article_id >= 900000
--Abfrage war erfolgreich durchgeführt: 3000 Zeilen, 97541 ms
Ausführungszeit.

SELECT * FROM query('ojthrlqvl');
--Gesamtlaufzeit der Abfrage: 224 ms. 0 Zeilen geholt.
SELECT * FROM query('ojthtestrlqvl');
--Gesamtlaufzeit der Abfrage: 156 ms. 0 Zeilen geholt.
```

**Listing 11-3: Funktionstest - PPSv2**

```
SELECT * FROM findArticle('confounding&expectations&mexico&capel');
--Gesamtlaufzeit der Abfrage: 131 ms.
--1 Zeilen geholt.

SELECT * FROM findArticle('confounding&expectations&mexico');
--Gesamtlaufzeit der Abfrage: 215 ms.
--1 Zeilen geholt.

SELECT * FROM findArticle('confounding&expectations');
--Gesamtlaufzeit der Abfrage: 294 ms.
--15 Zeilen geholt.

SELECT * FROM findArticle('confounding');
--Gesamtlaufzeit der Abfrage: 156 ms.
--27 Zeilen geholt.

SELECT * FROM findArticle('money');
--Gesamtlaufzeit der Abfrage: 45911 ms.
--4017 Zeilen geholt.

SELECT * FROM findArticle('exchange');
--Gesamtlaufzeit der Abfrage: 15603 ms.
--6926 Zeilen geholt.
```

**Listing 11-4: Performancetest - Tsearch2**

```
SELECT * FROM query_simple('confounding expectations mexico capel');
--Gesamtlaufzeit der Abfrage: 452 ms.
--1 Zeilen geholt.

SELECT * FROM query_simple('confounding expectations mexico');
--Gesamtlaufzeit der Abfrage: 172 ms.
--1 Zeilen geholt.

SELECT * FROM query_simple('confounding expectations');
--Gesamtlaufzeit der Abfrage: 156 ms.
--15 Zeilen geholt.

SELECT * FROM query_simple('confounding');
--Gesamtlaufzeit der Abfrage: 218 ms.
--27 Zeilen geholt.

SELECT * FROM query_simple('money');
--Gesamtlaufzeit der Abfrage: 49124 ms.
--4017 Zeilen geholt.

SELECT * FROM query_simple('exchange');
--Gesamtlaufzeit der Abfrage: 39765 ms.
--6926 Zeilen geholt.
```

**Listing 11-5: Performancetest – PPS**

```
SELECT * FROM query('confounding expectations mexico capel');
--Gesamtlaufzeit der Abfrage: 3993 ms.
--1 Zeilen geholt.

SELECT * FROM query('confounding expectations mexico');
--Gesamtlaufzeit der Abfrage: 265 ms.
--1 Zeilen geholt.

SELECT * FROM query('confounding expectations');
--Gesamtlaufzeit der Abfrage: 281 ms.
--15 Zeilen geholt.

SELECT * FROM query('confounding');
--Gesamtlaufzeit der Abfrage: 265 ms.
--27 Zeilen geholt.

SELECT * FROM query('money');
--Gesamtlaufzeit der Abfrage: 52696 ms.
--4017 Zeilen geholt.

SELECT * FROM query('exchange');
```

```
--Gesamtlaufzeit der Abfrage: 10142 ms.
--6926 Zeilen geholt.
```

**Listing 11-6: Performancetest – PPSv2**