Fachhochschule Aachen
Campus Jülich

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

# Masterarbeit

**Konzeption und Entwicklung einer Zitationsdatenbank zur Analyse bibliographischer Massendaten**

vorgelegt von Sebastian Schindler

Jülich, den 16. August 2012

| | |
|---|---|
| Fachbereich: | Medizintechnik und Technomathematik |
| Studiengang: | Technomathematik |
| Matrikelnummer: | 820620 |


Erstellt in der

---

## Zentralbibliothek

des

## Forschungszentrums Jülich

---

Diese Arbeit wurde betreut von:

1. Prüfer: Prof. Dr. rer. nat. Volker Sander
2. Prüfer: Dipl. Phys. Ing. Waldemar Hinz
3. Prüfer: M. C. Sc. Cornelia Plott

# Eigenständigkeitserklärung

Diese Arbeit ist von mir selbstständig angefertigt und verfasst. Es sind keine anderen als die angegebenen Quellen und Hilfsmittel benutzt worden.

| | |
|---|---|
| Ort, Datum | Sebastian Schindler |

# Inhaltsverzeichnis

# Kapitel 1

# Motivation und Einleitung

Den Erfolg und die Bedeutsamkeit der Arbeit wissenschaftlicher Forschungsinstitutionen miteinander zu vergleichen ist eine komplexe Herausforderung. Dieser und weiteren Aufgaben hat sich eine eigene Forschungsdisziplin, die Bibliometrie[1], gewidmet. Als Grundlage ihrer Untersuchungen bedienen sich die Bibliometriker hierbei der End-, Kommunikations- bzw. Zwischenprodukte eines jeden Wissenschaftlers: Den schriftlichen Publikationen. Um nicht nur die Quantität des wissenschaftlichen Outputs, sondern auch dessen Einfluss messen zu können, nutzt die Bibliometrie verschiedene Metriken, welche häufig unter anderem davon abhängig sind, wieviele andere Veröffentlichungen diese Arbeit zitieren. Zu diesem Zweck müssen alle Veröffentlichungen weltweit gesammelt, katalogisiert und über ihre Referenzen miteinander verknüpft werden. Dies haben sich einige Unternehmen zur Aufgabe gemacht. So wurden riesige Ansammlungen von Daten zusammengetragen und öffentlich oder kommerziell zugänglich gemacht.
Auch die Zentralbibliothek des Forschungszentrums Jülich befasst sich mit der Bibliometrie. Damit tiefgehende Analysen der wissenschaftlichen Arbeiten vorgenommen werden können, wurden mit dem Web of Science und Scopus die beiden maßgeblichen multidisziplinären Zitationsdatenbanken lokal angeschafft. Zwar steht zu diesen beiden Datenquellen auch ein öffentlicher Onlinezugriff zur Verfügung, dieser bietet aber eine begrenzte Möglichkeit, Suchanfragen durchzuführen. Zudem bietet dieser Zugang keine Funktiona-

lität für automatisierte Berechnungen von Zitationsmetriken.
Die Anwendung, in der die Daten lokal im Forschungszentrum archiviert und durchsuchbar gemacht werden, unterstützt zwar die Zählung der Zitationsrate, kann diese aber für eine solche Masse an Daten nicht schnell genug durchführen.
Um den Bibliometrikern vor Ort die Analyse einer solch immensen Datenmenge zu ermöglichen, befasst sich diese Masterarbeit mit der Konzeption und Entwicklung einer Datenbank, welche die Datenhaltung und Berechnung verschiedenster Zitationsmetriken erlaubt und außerdem Schnittstellen zu anderen analytischen Anwendungen bietet. Ein besonderes Augenmerk wird hierbei auf die Performanz gelegt, da die Bearbeitung von Massendaten stets eine zeitkritische Angelegenheit ist, Rechenzeit jedoch kostspielig ist.

Um einen Überblick über das Ziel dieser Arbeit zu geben, werden im zweiten Kapitel kurz relevante Handwerkszeuge der Bibliometrie skizziert.

Das dritte Kapitel befasst sich anschließend mit der Problemstellung dieser Arbeit und beinhaltet Analysen und Überlegungen, wie eine mögliche Lösung aussehen könnte.

Die konkrete Implementierung dieser Lösung wird im vierten Kaptitel erötert.

Abschließend wird im letzten Kapitel das Ergebnis zusammengefasst, bewertet sowie über diverse Erweiterungsmöglichkeiten diskutiert.

# Kapitel 2

# Grundlagen

## 2.1 Bibliometrie

### 2.1.1 Aufgaben der Bibliometrie

Die Bibliometrie ist eine Teildisziplin der Informetrie[2], und hat sich die „Anwendung mathematischer und statistischer Methoden zur Erklärung der Prozesse von schriftlichen Mitteilungen“ [4] zur Aufgabe gemacht. Mit ihrer Hilfe ist man in der Lage, Aussagen über die Menge bzw. die Zunahme des Wissens der Menschheit oder den Einfluss der Arbeit einer Forschungsinstitution zu treffen. Dabei bedient man sich an sowohl aus der Mathematik als auch aus der Informationswissenschaft stammenden Methoden[3].

Das wichtigste Handwerkzeug eines Bibliometrikers ist die sogenannte Zitationsanalyse. Grundlage ist die Häufigkeit, mit der ein bestimmter Artikel, Autor, Institut oder geographischer Ort[1] in anderen Arbeiten zitiert wird und wieviele andere Publikationen er selbst referenziert. „Zitate haben in der Wissenschaft ihre größte Bedeutung. Wissenschaftler sind stets darauf angewiesen, Arbeiten anderer Personen zu verwenden, damit etwa unnötige Wiederholungen eines Experiments verhindert werden. Der Wissenschaftler arbeitet sozusagen auf den Schultern eines Riesen (d.h. auf der Erfahrung

[1] Beispielsweise ein Land.

seiner vielen Vorgänger): Zum Beispiel wird im einleitenden Text einer Dissertation mit Zitaten belegt, welche Aspekte des Themas schon bekannt sind und welche Wissenslücken noch bestehen.“ [4].

Die Begriffe Referenz, Zitat und Zitation[2] müssen hierbei und im weiteren Kontext dieser Masterarbeit streng voneinander unterschieden werden. Zitat und Referenz sind die Anerkenntnis des Autors, dass er die Arbeit eines anderen Autoren als Quelle verwendet und diese zitiert (aktiv). Eine Zitation hingegen ist die Anerkenntnis für den Autor, der zitiert wurde (passiv).
Um Aussagen über die Sichtbarkeit und Nutzung der Publikationen der Forscher treffen und diese miteinander vergleichen zu können, stehen einer Zitationsanalyse eine Vielzahl unterschiedlicher Zitationsmetriken zur Verfügung. Diese Metriken sind von mehr Faktoren als nur der reinen Zitationsquantität abhängig. Dadurch ermöglichen sie es, den Einfluss von Ergebnissen von Forschergruppen beispielsweise verschiedener Größe oder aus unterschiedlichen Forschungsdisziplinen sinnvoll miteinander zu vergleichen.

### 2.1.2 Zitationsmetriken

#### Allgemein

Eine Zitationsmetrik ist ein Indikator, mit dessen Hilfe sich beschreiben lässt, wie sehr eine bestimmte wissenschaftliche Publikation von anderen Wissenschaftlern wahrgenommen wird.
Es wird zwischen unterschiedlichen Typen differenziert. Die wichtigsten sind die sogenannten grundlegenden, normalisierten und gewichteten Indikatoren[10].
Die richtige Metrik in der richtigen Situation zu verwenden ist Bestandteil einer Zitationsanalyse und somit Aufgabe der Bibliometrie.

[2]Nicht zu verwechseln mit den englischen Begriffen: reference, quote, citation (Reihenfolge identisch).

### Grundlegende Indikatoren

In die Kategorie der grundlegenden Indikatoren fallen die einfachsten Indikatoren, welche auf der reinen Zitationssanzahl beruhen. Oft wird eine durchschnittliche Zitierungquantität pro Zeiteinheit oder pro veröffentlichte Arbeit ermittelt.
Der weitverbreitetste Indikator dieser Kategorie ist der (Journal[3]) Impact Factor. Er bezieht sich auf ein bestimmtes Jahr und Journal und ermittelt die erhaltenen Zitationen innerhalb des Bezugsjahrs für alle Publikationen des Journals, die in den letzten beiden Jahren vor dem Bezugsjahr veröffentlicht wurden:

$$IF_{j,y} = \frac{C_{j^{y-1},y} + C_{j^{y-2},y}}{P_{y-1} + P_{y-2}} \tag{2.1}$$

wobei es sich bei $C_{j^{y-1},y}$ und $C_{j^{y-2},y}$ um die Anzahl der im Jahr $y$ erhaltenen Zitationen des Journal $j$ aller in den Jahren $y-1, y-2$ veröffentlichten Artikeln handelt. $P_{y-1} + P_{y-2}$ beschreibt die Anzahl der veröffentlichten und „zitierbaren“ Artikeln des Journal $j$ in den Jahren $y-1, y-2$[5, 6].
Da der Impact Factor nur von der Anzahl der Zitationen und veröffentlichten Artikeln abhängt, ist er ungeeignet dafür, Journals aus verschiedenen Disziplinen miteinander zu vergleichen, da beispielweise eine wissenschaftliche Arbeit aus dem Bereich Mathematik viel weniger Zitate als aus dem Bereich Chemie verwendet. Das Problem der Disziplinabhängigkeit trifft bei allen grundlegenden Indikatoren zu.

### Normalisierte Indikatoren

Das Prinzip der normalisierten Indikatoren ist es, auch unterschiedliche Disziplinen (wie der Name bereits vermuten lässt) mithilfe einer Normalisierung sinnvoll miteinander vergleichen zu können. Der Indikator bestimmt eine Zititierungsrate abhängig von der weltweit durchschnittlichen Zitationsrate seiner Disziplin. Diese Normalisierung kann a posteriori geschehen, wobei die Publikationen, deren Zitationsrate bestimmt werden sollen, normalisiert wer-

[3]Fachzeitschrift, regelmäßig erscheinende Sammlung von Artikeln zu einer bestimmten Fachrichtung.

den. Man bestimmt also zuerst die Zitationsrate, und normalisiert anschließend in Abhängigkeit der zu untersuchenden Publikationen (cited-site). Im Gegensatz dazu kann auch anhand jener Publikationen normalisiert werden, welche die zu betrachtende Publikationen zitieren. Hierbei handelt es sich dementsprechend um eine a priori Normalisierung, da die zu bestimmende Zitationsrate insofern normalisiert wird, weil das Endergebnis durch einzelne gewichtete Zitationswerte bestimmt wurde (citing-site)[7].
Ein weitverbreitetes Beispiel für die a priori Normalisierung ist das sogenannte Fractional Counting. Der sogenannte Fractional-counted Impact Factor ist wie folgt definiert:

$$fractionalIF = \frac{\sum_{\frac{1}{r_p}}}{P} \tag{2.2}$$

wobei $r_p$ die Anzahl der Referenzen der Publikation $p$ und $P$ die Anzahl aller zu betrachteten Publikationen darstellt[8, 9].
Dieser Indikator sorgt dafür, dass jede Zitation, die ein Dokument erhält, die Zitationsrate nicht um 1, sondern nur um $\frac{1}{r_p}$ erhöht. Dies führt also dazu, dass zitierende Publikationen mit vielen Referenzen weniger Wert sind. Auf diese Weise kann das oben beschriebene Problem, dass Artikel aus dem Bereich Chemie im Durchschnitt viel mehr Referenzen verwenden[10] als Artikel aus der Mathematik, gelöst werden. Angenommen, eine Publikation A aus dem Bereich der Chemie hätte 30, eine andere Publikation B aus dem Bereich der Mathematik hätte 5 Referenzen. Eine Zitation der Publikation B wäre demnach $\frac{1}{5}$ wert, eine der Publikation A hingegen nur $\frac{1}{30}$.

## 2.2 Bibliographische Daten

### 2.2.1 Allgemein

Um solche in 2.1 erwähnten Zitationsanalysen durchzuführen, bedarf es einer Sammlung an Publikationen, die über ihre Referenzen miteinander verknüpft sind.
Viele Bibliotheken verwenden bibliographische Daten, um Wissenschaftlern Recherchen in wissenschafltichen Veröffentlichungen zu ermöglichen. Dabei

handelt es sich um Informationen, die alle wichtigen Metadaten wie beispielsweise Autor, Publikationsjahr, Publikationsland, Institut, Abstract, ISSN oder sonstige Identifikatoren, nicht aber den eigentlichen Text des Artikels, enthalten. Eines der gängigsten Formate, das Bibliotheken zur Verwaltung von bibliographischen Datensätzen verwenden, ist MARC [4]. Einige Firmen bzw. Einrichtungen haben es sich zur Aufgabe gemacht, bibliographische Daten, die eine große Menge aller weltweit publizierten Paper abdeckt, zu sammeln und öffentlich oder kommerziell zugänglich zu stellen. Diese bibliographischen Daten verwenden die Bibliometriker als Grundlage ihrer Analysen.

### 2.2.2 Bibliographische Daten der Zentralbibliothek

Die Zentralbibliothek des Forschungszentrums Jülich betreibt ebenfalls bibliometrische Analysen. Zu diesem Zwecke wurden Daten aus zwei bekannten und umfangreichen Datensammlungen käuflich erworben.
Die erste Sammlung ist Scopus[12]. Dies ist eine Abstract- und Referenz-Datenbank, die Metadaten von insgesamt 44 Mio. Publikationen aus 18,500 Journals enthält. Bei den erworbenen Datensätze handelt es sich um 23 Mio. Dokumente aus den Jahren 1996 bis heute. Jede Publikation ist eindeutig mit einer sogenannten SGR-Nummer indentifizierbar. Das XML-Format, in denen sie ausgeliefert wurden, verwendet ein eigenes, von Scopus konzipiertes Schema (siehe 2.1).

[4] Machine-Readable Cataloging, siehe [13].

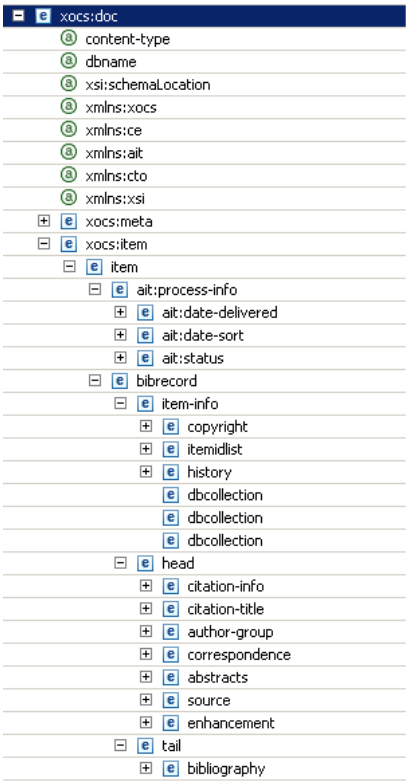



Abbildung 2.1: Beispiel eines Scopus Datensatzes im XML-Editor

Die zweite von der Zentralbibliothek beschaffte Datensammlung ist das Web of Science[11]. Diese vom ISI[5] (heute Thomson Reuters) erstellte Datenbank beinhaltet Metadaten von mehreren Millionen interdisziplinären Publikationen aus über 12.000 Journals. Auch hier gibt es einen eindeutigen Identifikator, den UT-Code. Die Auslieferung der Lokaldaten erfolgt durch eine vielzahl von Dateien, die jeweils duchschnittlich 30.000 Datensätze enthalten, dargestellt in einem von Thomson Reuters entwickeltem Format. Das Format entspricht keinem Standard wie beispielsweise XML oder JSON, sondern basiert auf einer nicht-hierarchischen Text-Struktur von auszeichnenden Tags (siehe 2.2).

[5]Institute for Scientific Information.

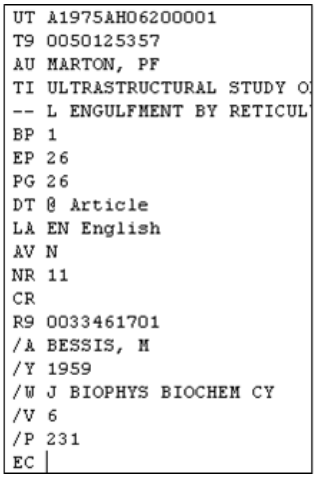



Abbildung 2.2: Beispiel eines Web of Sciences Datensatzes

Die Datensätze innerhalb einer dieser beiden Datenbanken sind teilweise mithilfe von eindeutigen Identifikatoren miteinander verlinkt. So kann man direkt zu den Referenzen und zitierenden Datensätzen jeder Publikation navigieren.
Zudem gibt es keine Garantie, dass die gelieferten Datensätze fehlerfrei sind. Diese Befürchtung wird dadurch bestätigt, dass einige der gelieferten Dateien Korrekturen für die anderen Datensätze beinhalten. Deshalb muss bei der Verarbeitung besonders auf Dubletten oder offensichtliche Fehler geachtet werden.

# Kapitel 3

# Lösungsentwurf

## 3.1 Problemstellung

Wie bereits in Kapitel 2.1 beschrieben, hat die Zentralbibliothek des Forschungszentrums Jülich Daten aus zwei bekannten und umfassenden bibliographischen Datenbanken erworben. Um diese Daten effektiv nutzen zu können, werden sie mithilfe von CDS Invenio indexiert und somit digital durchsuchbar gemacht. Bei CDS Invenio handelt es sich um eine vom CERN entwickelte open-source Applikation, die die Erstellung von digitalen Bibliotheken ermöglicht und als Suchmaschine fungiert[15].
Weil die Daten allerdings nicht nur durchsuchbar sein sollen, sondern auch für bibliometrische Analysen aufbereitet werden müssen, ist es für jeden Datensatz weiterhin notwendig, zu bestimmen, welche und wieviele Datensätze ihn zitieren. Eine solche Berechnung ist mittels Invenio zwar automatisiert durchführbar, aufgrund der hier verwendeten Datenmenge von vielen Millionen Datensätzen allerdings nicht ratsam, da sie mit einer solchen Masse nicht skaliert. Weiterhin bietet Invenio keine Unterstützung für die Vielfalt an Zitationsmetriken (siehe Abschnitt 2.1.2), die für eine umfangreiche Zitationsanalyse benötigt werden.
Um dieses Problem zu lösen, soll im Rahmen dieser Arbeit eine entsprechende Software implementiert werden, welche die Möglichkeit liefert, umfangreiche Zitationsanalysen auf allen bibliographischen Daten durchzuführen.

Diese Anwendung soll Invenio nicht direkt erweitern, sondern unabhängig davon agieren können.

## 3.2 Anforderung an die Funktionalitäten

Wie oben genannt, soll die Anwendung in der Lage sein, einen Bibliometriker bei Zitationsanalysen auf den vorhandenen Daten zu unterstützen. Dazu soll sie die Berechnung möglichst vieler verschiedener Zitationsmetriken beherrschen. Eine solche Analyse bezieht sich immer auf eine bestimmte Untermenge der Daten, weshalb die Möglichkeit bestehen muss, die zu analysierenden Daten zu selektieren.
Ferner sollte die Anwendung mit Invenio koppelbar sein, sodass die bereits zur Verfügung stehenden Features und Daten weiterverwendbar sind. Invenio bietet hierzu einige Schnittstellen an. Auch die zu implementierende Anwendung sollte Schnittstellen anbieten, um eine Erweiterbarkeit zu gewährleisten. So wäre beispielsweise die Koppelung mit einem anderen bibliographischen Tool denkbar.
Der besondere Knackpunkt der Aufgabenstellung ist jedoch die Performanz. Wie bereits in 2.1.2 angedeutet, ist die Berechnung solcher Zitationsmetriken unter Umständen eine sehr komplexe Angelegenheit. In Verbindung mit dem riesigen Pool an Datensätzen, die bearbeitet werden müssen, wird schnell klar, dass ein Augenmerk auf die Laufzeit gelegt werden muss. Die Bibliometriker müssen in der Lage sein, Analysen auch auf größeren Datenmengen durchzuführen und die Ergebnisse möglichst zeitnah zu erhalten. Ein Beispiel für eine solche laufzeitkritische Analyse wäre die Frage: „Wie oft wird die USA von Europa zitiert?“.
Eine letzte, abschließende Anforderung ist die Erweiterbarkeit im Sinne einer neuen Datenquelle. Momentan stehen Publikationen aus zwei bestimmten Datenbanken zur Verfügung. Da es sehr gut möglich ist, dass in Zukunft auch noch weitere Datenquellen hinzugezogen werden, sollte die Anwendung unabhängig vom Format der gelieferten Daten sein. Schon an den zwei bereits vorhandenen Datenbanken wird deutlich, dass bibliographische Daten in sehr

unterschiedlichen Formaten ausgeliefert werden können.
Zusammenfassend sollen also folgende Anforderungen erfüllt werden:

- Berechnung von Metriken zu selektierten Datensätzen
- Koppelung mit CDS Invenio
- Schnittstellen zu anderen Tools
- Laufzeit der Berechnung minimal halten
- Unabhängigkeit vom Eingabeformat einer Datenquelle

## 3.3 Strategien zur Erstellung einer Zitationsdatenbank

### 3.3.1 Vorüberlegungen

Wie bereits in 3.1 beschrieben, sollen die bibliographischen Daten hauptsächlich von Invenio verwaltet werden. Invenio nutzt eine Datenbank, die einen Indexer realisiert, und ist dadurch in der Lage, alle Datensätze schnell zu durchsuchen und jeden beliebigen Datensatz in verschiedenen Formaten anzuzeigen.
Die zu implementierende Anwendung soll ein externes Programm sein, dass ausschließlich die Berechnung der Zitationsanzahl sowie weiteren Zitationsmetriken übernimmt. Die Basis der Anwendung bildet eine Datenbank, weil nur ein professionelles Datenbankmangementsystem in der Lage ist, eine solche Masse an Daten performant und dennoch flexibel zu verwalten.
Die Performanz, die sich am Ende ergibt, setzt sich hauptsächlich aus drei Komponenten zusammen (siehe auch Abb. 3.1): Zum einen spielt natürlich der Server eine Rolle. Die verwendete Hardware, die Gesamtauslastung des System durch andere Anwendungen und auch die Konfiguration des Datenbankservers selbst. Zum anderen ist es sehr wichtig, wie das Schema der Datenbank strukturiert ist bzw. welche Einstellungen die Tabellen haben. Der dritte Faktor ist die Struktur der Abfragen.

Hierbei spielen die letzten beiden genannten Komponente die aller größte Rolle, da sie die größte Auswirkung auf die Gesamtperformanz haben. In diesem Kapitel sollen nur sowohl Server- als auch Schemaseitige Lösungansätze skizziert werden. Die Struktur der Abfrage wird anschließend im Kapitel 4 erläutert.

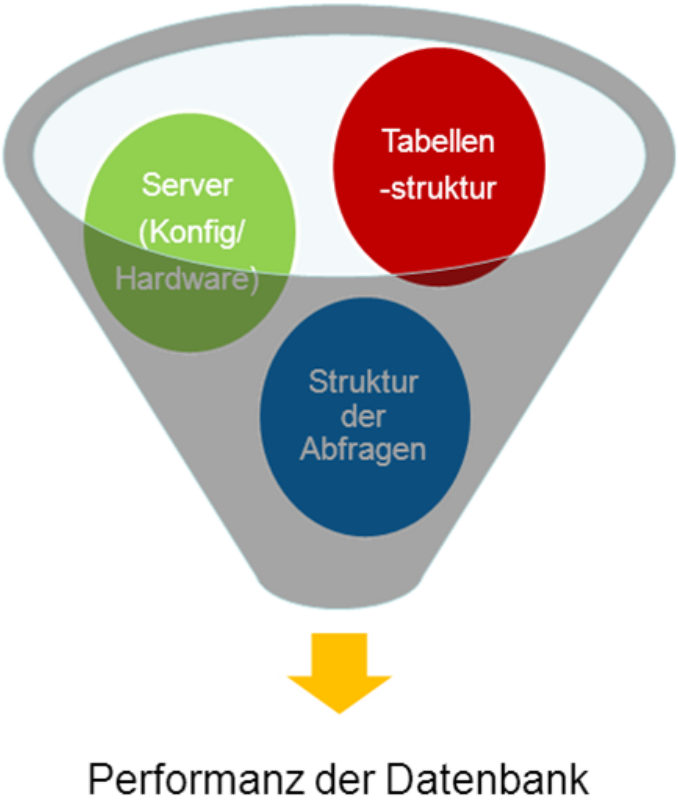



Abbildung 3.1: Die drei wichtigsten Komponente der Performanz einer Datenbankanwendung

### 3.3.2 Datenbankmanagementsystem

Der erste Schritt zur Erstellung der Zitationsdatenbank ist die Auswahl des Datenbankmanagementsystems. Hier steht man zwar allgemein vor einer sehr großen Auswahl, welche jedoch unter den Bedingungen dieses Projektes schnell getroffen ist: *MySQL*. Begründen lässt sich diese Entscheidung

wie folgt: Zum einen ist nicht gänzlich klar, unter welchem Betriebssystem das Datenbankmanagementsystem in ferner Zukunft laufen wird, weshalb plattformabhängige Datenbanken wie beispielsweise *MS SQL* nicht in Frage kommen. Da das Projekt möglichst ohne zusätzliche Kosten auskommen sollen, fallen auch Datenbanken wie beispielsweise Oracle weg. Weiterhin ist MySQL in der IT-Branche sehr verbreitet, was den Vorteil hat, das ausgiebige Dokumentationen und eine große Community konsultiert werden können. Zum anderen stehen Erfahrungsberichte zur Verfügung, aus denen hervorgeht, dass MySQL auch Tabellen verwalten kann, die mehrere hundert Millionen Zeilen enthalten. Letztendlich spielt auch die Erfahrung, die die Mitarbeiter der Zentralbibliothek mit MySQL haben, eine große Rolle, denn auch Invenio nutzt eine MySQL-Datenbank. Die Administratoren und zukünftige Entwickler hätten also keine Einarbeitungszeit, um einen MySQL-Server zu verwalten und das Schema zu erweitern.

### 3.3.3 Datenbank-Schema

Aufgrund der Datenmenge, mit der gearbeitet wird, muss das Schema dieser Zitationdatenbank möglichst einfach gehalten werden. Deshalb ist es besonders wichtig, das richtige Gleichgewicht zwischen Nutzen und Einfachheit zu finden.

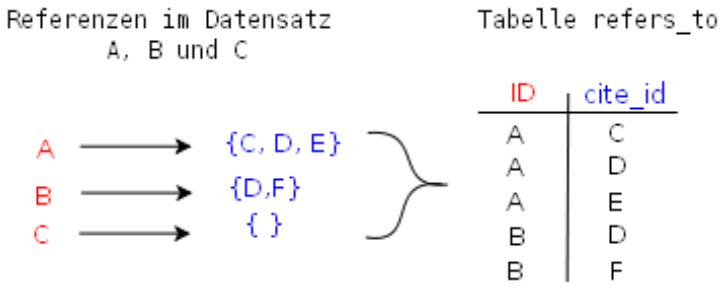



Abbildung 3.2: Speicherung der Referenzen in der DB

Die Hauptaufgabe der Zitationdatenbank wird es sein, zu jedem Datensatz zu speichern, welche anderen Datensätze er referenziert. Hierzu muss

möglichst schnell bestimmt werden können, wieviele und welche anderen Datensätze ihn zitieren bzw. er zitiert. Konkret wird also eine Verknüpfung eines Datensatzes mit mehreren anderen abgebildet (siehe hierzu Abbildung 3.2). Um diese Verknüpfung zu realisieren, macht es Sinn, einen Identifikator zu verwenden, der jeden Datensatz eindeutig bestimmt. Eine solche interne ID könnte leicht mithilfe des Datenbankmanagementsystems erstellen werden, da die Datensätze allerdings auch nach außen hin eindeutig sein sollten, kann der der bibliographischen Daten spezifische Identifikator verwendet werden. Auf diese Weise spart man sich später eine zusätzliche Verknüpfung, da die interne ID nicht in die echte aufgelöst werden muss.
Weiterhin bietet es sich an, einige zusätzliche Informationen zu jedem Datensatz zu speichern, damit diverse Metriken direkt auf der Datenbank und somit schnell berechnet werden können. So ist beispielsweise das Publikationsjahr eine wichtige Größe, die von sehr vielen Metriken genutzt wird.

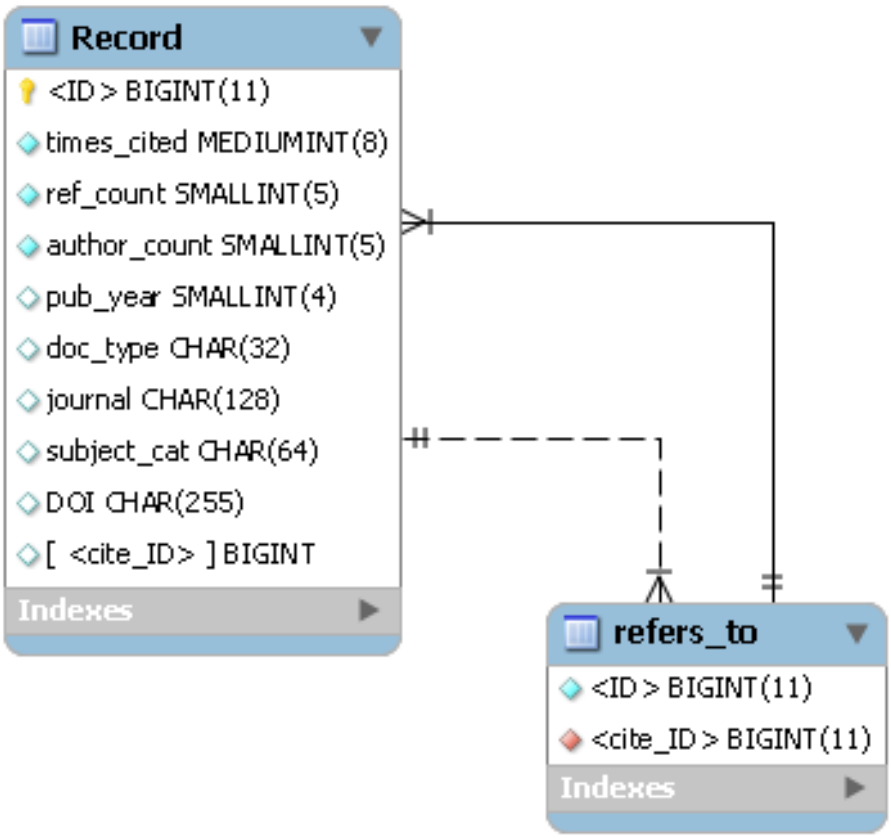



Abbildung 3.3: Allgemeines ER-Diagramm des Schemas

So lässt sich das Schema in zwei Teile unterteilen: Der Erste, der eine Untermenge der Metadaten von jedem Datensatz speichert und ein zweiter, der zitierte Datensätze miteinander verknüpft. Betrachtet man diese Ausgangssituation nun im Entity-Relationship-Modell, stellt der Teil, in dem die Datensätze gespeichert werden, die Entität *Record* dar. Zitate können dann als Referenz eines Records auf einen anderen Record gesehen werden, d.h. als rekursive n-1-Beziehung zwischen zwei Records. Da eine solche rekursive n-1-Beziehung in einer eigenen Tabelle gespeichert werden muss, ergeben sich somit zwei Tabellen: Die Tabelle *Record* und die Tabelle *refers_to*. Natürlich könnte man die n-1-Beziehung auch *cited_by* nennen, da sie aus zwei Spalten bestehen muss, nämlich eine, die den Akteur angibt, also *wer* wird zitiert bzw. referenziert, und die andere, die den Empfänger angibt, sprich *wen* bzw.

*von wem* wird sie zitiert bzw. referenziert. Ob Datensatz A nun Datensatz B referenziert oder Datensatz B von A zitiert wird, spielt für die Speicherung keine Rolle, sondern nur, dass es eine Zeile in der Tabelle gibt, in der A und B gemeinsam vorkommen.
Wie in 3.2 beschrieben, soll die Anwendung, welche die Datenbank erstellt und nutzt, mit verschiedenen Datenquellen gefüllt werden können. So variieren gewisse Inhalte des Schemas, beispielsweise Datentypen, Datengröße und Namen der Felder, da sie bis zu einem bestimmtem Grad dynamisch an die Datenquelle angepasst werden sollen. Ein allgemeines ER-Diagramm des Datenbankschemas ist in Abbildung 3.3 ersichtlich. Hierbei bedeuten spitze Klammern, dass dort der konkrete Name eingefügt wird. Eckige Klammern sollen optionale Attribute (je nach Datenquelle) andeuten.

### 3.3.4 MySQL-spezifische Optimierungen

#### Benchmarking

Die Erstellung der Zitationsdatenbank ist ein umfangreiches Projekt. Deshalb ist es notwendig, sich bereits im Vorfeld Gedanken über die Performanz zu machen. Die folgenden Abschnitte dieses Kapitels werden sich mit einigen Optimierungsmöglichtkeiten beschäftigen, da man mit einer Standardkonfiguration und schlecht geplanten Strukturen schnell an die Grenzen der von MySQL bzw. der Hardware stößt.
Um die Auswirkung dieser Optimierungen zu validieren, empfiehlt es sich, die Änderungen mithilfe von Benchmark-Tests und einer Monitoring-Software zu überprüfen.

Eine gute Vorgehensweise ist es, zu ermitteln, welche konkreten SQL-Abfragen in der fertigen Anwendungen hauptsächlich genutzt werden sollen, und diese anschließend zu testen. Wichtige Indikatoren sind hierbei natürlich die Laufzeit der Abfrage, aber auch Systemressourcen wie Cache-Nutzung, I/O-Zugriffe auf die Festplatte sowie die Cache-Hitrate. Diese Informationen werden im Rahmen dieser Arbeit mit dem MySQL-Monitoring-Tool *MONyog*[16] ermittelt. So werden verschiedenste Konfigurationen und Ein-

stellungen miteinander verglichen, um letztendlich eine optimale Leistung erzielen zu können. Wichtig ist es hierbei, dass sich Änderungen auf manche Typen von Abfragen positiv, auf anderen hingegen negativ auswirken können, weshalb grundsätzlich jede Use-Case-Abfrage bei jeder Änderung getestet werden muss. Der Query-Cache von MySQL, welcher die Ergebnisse der letzten Anfragen speichert, um bei erneuter Anfrage schneller reagieren zu können, muss unbedingt ausgeschaltet sein, um verfälschte Ergebnisse vorzubeugen.

Weil ein Benchmark mit den gesamten Daten ein sehr langwieriger Prozess ist, da sowohl die Tests als auch die Änderungen am Schema sehr zeitintensiv sein können, wird eine Testumgebung verwendet. Bei dieser handelt es sich um eine in sich konsistente, aber künstlich generierte Sammlung von 6 Millionen Datensätzen. Jeder Datensatz hat einen hochgezählten Integer-Wert als ID, Referenzen auf 20 zufällig ausgewählte IDs, sowie einen erfundenen, aber von der Länge her realistischen Wert jeder übrigen Spalte. In Abbildung 3.4 sind die wichtigsten Abfragen, sortiert nach ihrer Priorität, aufgelistet:

| SQL-Abfrage | Verwendung | Priorität |
|---|---|---|
| **SELECT** ID **FROM** refers_to **WHERE** cite_id = 'x' | Alle Records abrufen, die Record *'x'* zitieren | hoch |
| **SELECT** * **FROM** Record **WHERE** ID = 'x' | Alle Daten des Records *'x'* abrufen | hoch |
| **SELECT** count(cite_id) **FROM** refers_to **WHERE** ID = *'x'* | Zitationszahl des Record *'x'* bestimmen | mittel |
| **INSERT INTO** *Record* **VALUES** (*'a','b','c','d','e','f'*) | Einladen eines Records | gering |
| **INSERT INTO** refers_to **VALUES** (*'a','b'*) | Einladen einer Referenz | gering |
| **UPDATE TABLE** x **SET** (a='x', b='y', ...) | Updaten einer Tabelle | gering |

Abbildung 3.4: Überblick über wichtigste Use-Case-Abfragen

### Indizes

Das oben beschriebene Schema ist zwar aus Sicht der relationalen Datenbank-Planung vollständig, es fehlen allerdings noch MySQL-spezifische Konfigurationen, die in diesem Abschnitt diskutiert werden sollen.
Die Datenbank soll später dazu verwendet werden, zu einer gegeben Liste von Datensätzen (repräsentiert durch ihre IDs) zu bestimmen, wie viele und welche andere Datensätze sie zitieren und mit dieser Anzahl gegebenenfalls in einem zweiten Schritt eine Zitationsmetrik zu bestimmen. Zu diesem Zwecke muss der Zugriff auf die Felder *ID* und *cite_id* besonders schnell durchführbar sein. Um dies sicherzustellen, ist die Verwendung eines Indexes auf diesen Feldern sinnvoll.
Ein Index ist eine Art sortierte Look-Up-Table, die die Datenbank intern erstellt um einzelne Werte gezielt finden zu können. Ohne Index wird ein

Scan auf der kompletten Tabelle durchgeführt, was zu einer Komplexität von $O(n)$ führt, wobei $n$ für die Anzahl aller in der Tabelle gespeicherten Zeilen steht. Die Nutzung eines Indexes ist also sinnvoll, um die Performanz einer Datenbankabfrage zu steigern. Ein weiterer Grund einen Index zu verwenden, stellen eindeutige Spalten bzw. Schlüssel dar. MySQL erstellt und nutzt automatisch einen Index, um Dubletten schnell zu erkennen und somit die Eindeutigkeit des Wertes innerhalb einer Tabelle zu gewährleisten.

MySQL bietet mehrere verschiedene Arten von Indizes: B-Baum, Hash-Map oder in neueren Version (5.x) auch R-Baum.
B-Bäume haben den Vorteil, dass sie besonders schnelle Suchen nach *min()* / *max()* und Bereichen erlauben. So würde beispielsweise folgendes SQL-Statement von einem B-Baum-Index auf der Spalte *nachname* profitieren:

```
SELECT * FROM mitarbeiter WHERE nachname
BETWEEN 'Schroeder' and 'Schmitz'
```

MySQL würde die Werte 'Schroeder' und 'Schmitz' im Baum suchen und alle dazwischen liegenden Spalten ebenfalls als Treffer ausgeben. Die Suche eines einzelnes Wertes hat hierbei eine Komplexität von $O(log(n))$, insgesamt hätte diese Abfrage also die Laufzeit $2*O(log(n))$. Verglichen mit einem kompletten Tabellenscan ist also eine deutliche Performanzsteigerung erkennbar.
Die Nutzung des zweiten Index-Typen, der Hash-Map, bietet im Normalfall eine Komplexität von $O(1)$ pro Wert, der nachgeschlagen werden muss. Zwar erscheint dies auf dem ersten Blick schneller als der B-Baum, so ist es aber beispielsweise bei der obigen Intervallsuche von Nöten, dass jeder im Intervall enthaltene Wert einzeln nachgeschlagen wird. Somit resultiert die Beispielabfrage also insgesamt in einer Laufzeit von $m * O(1)$ bei gegebener Intervallgröße $m$ .
R-Bäume werden hauptsächlich für räumliche oder n-Dimensionale Daten genutzt und werden deshalb an dieser Stelle vernachlässigt.
Nachdem jetzt die Vorteile eines Indexes erläutert wurden, muss vollständigerweise auch auf die Nachteile eingegangen werden. Jeder Index muss persistent als Datei auf der Festplatte gespeichert werden, was unter Umständen

viel Speicherplatz benötigt. Weiterhin verlangsamt die Anwesenheit eines Indexes das Schreiben von Daten ungemein, da (normalerweise) nach jeder eingefügten Zeile nicht nur die Tabelle selbst, sondern auch der Index aktualisiert werden muss. Die Planung von Indizes ist also ein Balanceakt zwischen Lesegeschwindigkeit, Speicherplatz und Schreibgeschwindigkeit, weshalb immer mit Bedacht entschieden werden sollte.
Wendet man dieses theoretische Wissen nun auf den konkreten Fall der Zitationdatenbank an, scheint der Hash-Map-Index die beste Wahl zu sein, da momentan in keinem Use-Case eine Intervall- oder Min/Max-Suche vorgesehen ist. Da die Datensätze bei der bisherigen Planung ausschließlich über ihre *ID* bzw. *cite_id* selektiert werden, sollte ein Index über diese Felder in beiden Tabellen ausreichend sein. Für den Primärschlüssel *ID* der Tabelle *Record* wird bereits automatisch ein eindeutiger Index erzeugt. Um mögliche Fehler in den einzuladenden Daten zu vermeiden, durch die Dubletten entstehen könnten, wäre es sinnvoll, einen weiteren eindeutigen Index für die Tabelle *refers_to* zu verwenden. Dieser sollte dann gewährleisten, dass keine Referenz doppelt abgespeichert wird, was im späteren Verlauf die Zitationsrate verfälschen würde. Hierzu kann ein mehrspaltiger eindeutiger Index über das Tupel *(ID, cite_id)* verwendet werden.
Bei mehrspaltigen Indizes ist die Reihenfolge der Definition zu beachten. So können nur Abfragen, in denen nach Spalte $col_i$ selektiert wird, vom Index $(col_1, col_2, ..., col_n)$ profitieren, wenn auch $col_1$ bis $col_{i-1}$ (mit $1 \leq i \leq n$) in die Selektion der WHERE-Klausel einbezogen werden. Das bedeutet für diesen Fall, dass zwar die Abfrage

```
SELECT * FROM refers_to WHERE id=x
```

den Index nutzt, folgende Abfrage jedoch nicht:

```
SELECT * FROM refers_to WHERE cite_id=x
```

Aus diesen eben beschriebenen Gründen benötigt die Tabelle *refers_to* einen zweiten Index auf der Spalte *cite_id*.
Indizes, die aus mehreren Komponente bestehen, stellen in manchen Situationen einen Sonderfall dar. Im Normalfall schlägt MySQL den gesuchten

| | **Record** | **refers_to** |
|---|---|---|
| **1. Index** | Primary Key *ID* | Unique *(ID, cite_id)* |
| **2. Index** | *cite_id* | *(cite_id, ID)* |

Tabelle 3.1: Übersicht über verwendete Indizes

Wert im Index nach, ermittelt die Spalten, in denen der Wert auftaucht, öffnet die eigentliche Tabelle, springt an die entsprechende Stelle und liest dort die gewünschten Ergebnisse aus. Liegt aber ein Index mit mehreren Komponenten vor, wie es beim eben beschriebenen *(ID, cite_id)*-Index der Fall war, muss folgende Abfrage nicht auf die eigentliche Tabelle zugreifen:

```
SELECT cite_id FROM refers_to WHERE ID=x
```

Die *cite_id*, welche abgefragt wird, kann direkt im Index ermittelt werden. Deshalb kann der langsame Zugriff auf die Tabelle vermieden werden, was einen deutlichen Geschwindigkeitsvorteil aufweist.
Um diesen Vorteil sowohl für die Abfrage der Referenzen als auch für die Bestimmung aller zitierenden Datensätze zu erhalten, wird der Index *(cite_id, ID)* als zweiter Index eingeführt. Das zusammengefasste Endresultat dieses Abschnittes ist der Tabelle 3.1 zu entnehmen.

### Datenbank-Engine

MySQL bietet eine Vielzahl verschiedener Speicher-Engines bzw. Tabellentypen an. So gibt es Tabellen vom Typ *Memory*, die vollständig in den Arbeitsspeicher abgelegt werden, die eher experimentellen Engines *Example* und *Blackhole*, eine Engine die nur auf Remote-Datenbanken arbeitet sowie die *NDB Cluster*-Engine, die Verwendung in MySQL-Cluster-Architekturen findet. Alle diese Tabellen-Typen sind jedoch für die geplante Zitationdatenbank unbrauchbar, da der benötigte Speicherplatz der Tabellen voraussichtlich zu groß für den vorhandenen Arbeitspeicher wird, nur ein Server vorhanden ist und somit keine verteilte Cluster-Nutzung vorliegt und die experimentellen Engines nur für kleinere Tests geeignet sind. Eine für das Pro-

jektvorhaben interessantere Datenbank-Engine hingegen stellt *Archive* dar. Dieser Tabellentyp ist in der Lage, bei wenig Speicherverbrauch eine große Menge an Daten zu verwalten. Leider ist es nicht möglich, einen Index anzulegen, was auch diese Engine ungeeignet macht. Aus diesem Grund muss die Wahl zwischen den beiden weit verbreitetsten Typen, *MyISAM* und *InnoDB*, getroffen werden[17]. Der größte Unterschied dieser beiden Engines ist, dass nur *InnoDB* transaktionssichere Tabellen bereitstellt. Nach eigener Aussage der MySQL-Entwickler ist die *MyISAM*-Engine dafür in der Lage, Daten sehr schnell zu speichern und abzurufen [21] .
Da die Performanz der Anwendung an erster Stelle steht, fällt die Entscheidung auf die *MyISAM*-Storage-Engine, was bedeutet, dass auf Transaktionssicherheit verzichtet wird. Dieser Verzicht ist allerdings leicht verkraftbar, weil ein Absturz des Servers während des Einladens der Metadaten von der zu erstellenden Anwendung erkannt und manuell oder automatisch wiederholt werden kann. Ein weiterer, schwerwiegenderer Schwachpunkt ist jedoch, dass sowohl für *MyISAM* als auch für *InnoDB* zumindest bei der hier verwendeten MySQL-Version 5.1 kein Hash-Index zur Verfügung steht.

### Partitioning

Ein zusätzlicher, fortschreitender Optimierungsansatz stellt das sogenannte Partitioning dar. Bei Tabellen, die eine gewisse Größe annehmen, ist es manchmal ratsam, diese Tabelle nach bestimmten Kriterien in mehrere kleineren Tabellen zu unterteilen. MySQL bietet die Möglichkeit, diese Aufteilung automatisch und für den Benutzer unsichtbar durchzuführen. So könnte man beispielsweise eine Tabelle, die Artikel katalogisiert, so partitionieren, dass die Artikel nach ihrem Erscheinungsjahr aufgeteilt werden. Das bedeutet, dass für jedes Jahr, das in der Artikelsammlung existiert, intern eine eigene Tabelle, sprich eine eigene Datei, angelegt wird. Dieses Vorgehen weißt einige Vorteile auf. Zum Einen kann MySQL bei Abfragen, die in der *WHERE*-Klausel das Jahr einschränken, nur die entsprechende Partition ansteuern und so direkt eine Masse an Zeilen ausschließen. Zum Anderen können Aggregatfunktionen wie *sum()* oder *count()* gut parallelisert werden.

Stehen sogar mehrere Festplatten zu Verfügung, können die Partitionen auf die Platten verteilt und die Parallelisierung somit noch besser ausgenutzt werden.

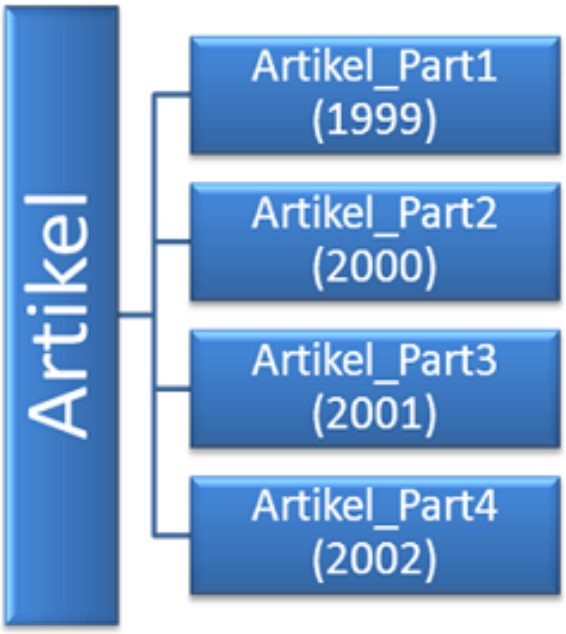



Abbildung 3.5: Partitionierung der Tabelle „Artikel“ nach dem Jahr

Möchte man Tabellen erstellen, deren Speicherplatz die maximale Speichermenge pro Datei des Betriebssystems überschreitet, ist man sogar gezwungen, eine Partitionierung zu verwenden.
Beim folgenden Benchmark, bei dem jeweils 5000 IDs als Eingabe dienen, soll getestet werden, ob eine Partitionierung der Tabelle *Record* bzw. *refers_to* durch Aufteilen der IDs Geschwindigkeitsvorteile vorweisen. Die Aufteilung geschieht hierbei durch einen Hash-Wert, der die Datensätze auf 20 Partitionen verteilt. Als Laufzeit ist jeweils der Durchschnitt von 5 einzelnen Messungen angegeben. Von einer Untersuchung der SQL-Abfragen mit geringer Priorität wird abgesehen. In der benutzten MySQL Version 5.1.61 ist Partitionierung ein neues, sich in der Testphase befindendes Feature und ist deshalb mit Vorsicht zu verwenden. Welche Algorithmen und Optimierungs-

| **Abfrage** | **Zeit in s ohne Partitioning** | **Zeit in s mit Partitioning (*Record*)** | **Zeit in s mit Partitioning (*refers_to*)** |
|---|---|---|---|
| Alle Records abrufen, die Record 1-5000 zitieren | 1.15 | 2.47 | 1.81 |
| Alle Daten der Records 1-5000 abrufen | 1.32 | 1.36 | 1.39 |
| Zitationszahl der Records 1-5000 bestimmen | 1.07 | 1.11 | 1.05 |

Tabelle 3.2: Benchmark mit und ohne partitionierten Tabellen

mechanismen MySQL intern wirklich verwendet, bleibt unklar. Betrachtet man die Ergebnisse des Benchmarks (siehe Tabelle 3.2), scheint es so, als ob Partitionierung im konkreten Fall entweder keine Veränderung oder gar eine Verschlechterung der Laufzeit bewirkt. Welche Gründe hinter diesem Verhalten liegen, erschließt sich leider nicht. Messfehler sind trotz mehrfacher Ausführung und anschließender Mittelung der Ergebnisse möglich.
Von der Verwendung von Partitionierung innerhalb der Zitationsdatenbank wird vorerst Abstand genommen. Dennoch sollte dieses Feature im Auge behalten werden, da Parallelisierung der Datenbank und Installation einer neueren MySQL-Version durchaus denkbare Schritte sind, um die Optimierung der Zitationsdatenbank zu verfeinern. Unter anderen Umständen könnte Partitionierung durchaus nützlich sein.

### Sonstiges

Abschließend zu den bereits genannten Optimierungsmöglichkeiten gibt es noch einige weitere Aspekte, die man beachten kann. Beispielsweise bietet MySQL die Möglichkeit, die Länge aller Zeilen in einer Tabelle auf einen konstanten Wert festzulegen, indem die Länge jeder Spalte festgelegt wird. Speichert man eine Zeichenkette, werden so viele Leerzeichen angehangen, bis ihre Länge der festgelegten Spaltenlänge entspricht. Das verbraucht zwar in viele Fällen deutlich mehr Speicherplatz, MySQL ist dafür aber in der Lage, effizientere Algorithmen beim Ein- und Auslesen zu verwenden, da jede Zeile gleich lang ist.
Ein Benchmark, bei dem jeweils 5000 IDs als Eingabe dienen, soll validieren,

| Abfrage | Zeit in s mit fester Spaltenlänge | Zeit in s mit dynamischer Spaltenlänge |
|---|---|---|
| Alle Records abrufen, die Record 1-5000 zitieren | 1.15 | 1.36 |
| Alle Daten der Records 1-5000 abrufen | 1.32 | 1.57 |
| Zitationszahl der Records 1-5000 bestimmen | 1.07 | 1.25 |

Tabelle 3.3: Benchmark mit und ohne fester Spaltenlänge

dass die feste Spaltenlänge performanter ist. Die Messwerte in Tabelle 3.3 zeigen deutlich, dass die Laufzeit bei Fester Spaltenlänge verringert wird.

### 3.3.5 Datenbank-Server

#### Hardware

Um die gewaltige Menge an Daten verarbeiten zu können, wird nicht nur eine Menge Festplattenspeicher, sondern auch viel Rechenleistung benötigt. Zur Durchführung des Projektes steht deshalb ein leistungstarker Server, auf dem nur das Betriebssystem, der MySQL Server und die zu entwickelnde Anwendung läuft, zur Verfügung. Der Vollständigkeit halber werden in Tabelle 3.4 die Eckdaten dieses Servers aufgelistet.
Da es sich bei Datenbankzugriffen hauptsächlich um Zugriffe aufs Dateisystem handelt, ist die Lese- bzw. Schreibgeschwindigkeit der Festplatte eine wichtige Größe. Dem Server stehen zwei Festplatten zur Verfügung, welche in einem Raid 0 Verbund zusammengeschlossen werden, um die Zugriffszeit des Systems bestmöglichst zu steigern.

#### Konfiguration

Die Anpassung der Konfiguration stellt den letzten Feinschliff der Optimierung dar. Häufig wird die Auswirkung einer umfangreichen Anpassung überschätzt. Die Struktur der Daten, das Schema und die Formulierung der Abfragen tragen den größten Teil dazu bei, dass die Datenbank performant arbeitet. Trotzdem soll die Konfiguration des Servers nicht gänzlich ver-

| Kenngröße | Wert |
|---|---|
| **Beschreibung** | Dell PowerEdge R610 |
| **CPU** | 12x Intel Xeon R5645 @ 2.40GHz |
| **Arbeitsspeicher** | 64GB |
| **Festplatte** | 2x PERC H700 1 TB (Raid 0) |
| **OS** | Redhat Enterprise Linux Server 6.2 |

Tabelle 3.4: Systemspezifikation des verwendeten Servers

nachlässigt, sondern in diesem Kapitel kurz skizziert werden.
MySQL bietet eine Vielzahl von Parametern, die frei einstellbar sind. Dementsprechend kompliziert ist auch die „optimale“ Konfiguration zu finden. Viele Werte haben untereinander Wechselwirkungen, sodass sich die Änderung einer Variable auch auf andere Größen auswirken kann. Die Erstellung einer optimalen Konfiguration geschieht am sinnvollsten iterativ. So werden konkrete Abfragen ausgesucht, die gezielt optimiert werden sollen, indem man nach und nach diverse Parameter verändert, von denen man sich eine Verbesserung verspricht und die Zeit misst, die diese Abfragen benötigen. Hierbei sollte im besten Fall immer nur ein Parameter gleichzeitig varriert werden[17]. Die aktuelle Belegung lässt entweder durch absetzen des SQL-Befehls *SHOW VARIABLES* oder durch einen Blick in die Konfig-Datei *my.cnf* in Erfahrung bringen. Auch der Befehl *SHOW STATUS* gibt Aufschluss über mögliche Engpässe. Er listet eine Reihe wichtiger Werte wie Anzahl langsamer Abfragen, abgebrochene Verbindungen, offene Tabellen oder die Menge an freiem Cache auf.
Beim konkreten Beispiel der Zitationsdatenbank müssen einige Parameter nicht betrachtet werden. Beispielsweise alle Variablen, die sich auf andere Engines als MyISAM beziehen, sind uninteressant. Weiterhin werden zur Laufzeit nie viele Verbindungen gleichzeitig aktiv sein, da die Datenbank darauf ausgelegt ist, dass maximal ein oder zwei Bibliometriker parallel

| Parameter | Bedeutung | gewählter Wert |
|---|---|---|
| *key_buffer_size* | Größe des Caches, der Indizes speichert | 16GB (max.) |
| *myisam_block_size* | Seitengröße des Caches | 4 kB |
| *tmp_table_size* | Speicher für temp. Tabellen | 1042 MB |
| *max_heap_table_size* | Speicher für Heap-Tabellen | 1024 MB |
| *tmpdir* | Pfad, an dem temp. Tabellen aufs Dateisystem gespeichert werden | n.A. |

Tabelle 3.5: Überblick über für die Zitationsdatenbank wichtige Parameter

Berechnungen darauf ausführen.
Die wichtigsten Parameter, die man verändern kann, sind die Größen der verschiedenen Caches. Bei der Entwicklung stehen jegliche Server-Ressourcen nur für MySQL zur Verfügung, weshalb nur genügend Arbeitsspeicher für das Betriebssystem berücksichtigt werden muss. Der Rest kann MySQL komplett zur Verfügung gestellt werden.
Der wichtigste Parameter ist *key_buffer_size*. Er legt die Menge an Arbeitsspeicher fest, die MySQL maximal reservieren darf, um Indizes in den Cache zu laden. Je kleiner der Wert, desto öfter muss das Betriebssystem Swappen, wodurch Verzögerungen entstehen. Der Wert dieser Variable ist jedoch bei MySQL 5.1 auf 16GB begrenzt. Empfohlen wird ein Wert, der zwischen 25% und 50% des gesamten zur Verfügung stehenden Arbeitsspeicher liegt[17].
Einen weiteren Parameter, den man unbedingt anpassen sollte, stellt *myisam_block_size* dar. Dieser stellt die Seitengröße des MySQL-Servers dar und sollte auf die Seitengröße des Betriebssystems angepasst werden, um die Interaktion zwischen MySQL und Betriebssystem zu optimieren, indem sogenannte *Read-Arround-Writes* vermieden werden [17]. Der Server, auf dem die Zitationsdatenbank läuft, benutzt eine Seitengröße von 4kB.

Bei großen Tabellen, wie es bei dieser Anwendung der Fall ist, sollte man auch die Steuerung von temporären Tabellen im Auge behalten. Hat eine Abfrage ein sehr großes Ergebnis oder eine große Menge an Daten, die sortiert werden soll, legt MySQL eine temporäre Tabelle im RAM an. Ist die maximal Größe der temporären Tabelle, konfigurierbar durch

die Variablen *tmp_table_size* und *max_heap_table_size*[1], erreicht, wird die Tabelle in eine Datei auf das Dateisystem geschrieben, was in langsamen I/O-Prozessen resultiert. Anhand der Status-Variablen *Created_tmp_tables* und *Created_tmp_disk_tables* lässt sich dieser Effekt feststellen. Um ihn zu vermeiden, sollte *tmp_table_size* möglichst groß gewählt werden. Da die Zitationsdatenbank kein typisches Multi-User-System ist, sondern eher nur einen Nutzer erwartet, der die Datenbank mit einem Thread gleichzeitig beschäftigt, kann die Variable *max_tmp_tables* auf einen kleinen Wert wie beispielsweise 8 gestellt werden. Leider kann die *tmp_table_size* nicht so hoch eingestellt werden, dass auch die größte temporäre Tabelle in den Cache passt, da zwar viel, aber für diese Datenmenge nicht genug Arbeitsspeicher vorhanden ist. Deshalb muss auch darauf geachtet werden, dass temporäre Dateien auf einer Partition bzw. Festplatte geschrieben werden, die stets genügend Speicherplatz bereitstellt. Vor allem beim Warten der Tabellen mittels *OPTIMIZE* und *ANALYZE* werden die kompletten Tabellen in einer tempörären Datei gespeichert. Der Pfad den MySQL hierfür wählt kann mit der Variable *tmpdir* gesteuert werden.

[1]Da temporäre Tabellen vom Typ „Heap“ sind, muss die *max_heap_table_size* ebenfalls erhöht werden.

# Kapitel 4

# Implementierung

## 4.1 Verwendete Requisiten

### 4.1.1 Datenbankmanagementsystem

Als Datenbankmanagementsystem wurde MySQL in der Version 5.1.61 verwendet. Diese Entscheidung wird in Kapitel 3.3.2 begründet.

### 4.1.2 Programmiersprache

Der im Rahmen dieser Arbeit implementierte Code ist in Python (Version 2.7) geschrieben. Python ist eine moderne und beliebte Skriptsprache, die sich durch Unterstützung von objektorientierter Programmierung auch für größere Projekte eignet. Zudem soll die Zitationsdatenbank zusammen mit Invenio auf einem Server laufen, und da Invenio auch in Python programmiert ist, bietet sich diese Sprache an. Hierdurch bleiben die Abhängigkeiten des Gesamtsystem übersichtlich.

### 4.1.3 Datenbanktreiber

Um von Python auf MySQL zuzugreifen, wird das Toolkit *SQL-Alchemy* [14] in Version 0.7 verwendet. Dabei handelt es sich um einen Wrapper für alle gängigen Datenbankentreiber, der objekt-relationales Mapping beherrscht.

So stehen bei Bedarf Objekte und Klassen zur Verfügung, die Datenbanken, Tabellen, Spalten usw. darstellen und Zugriff auf diese gewähren. Obwohl die Hersteller versprechen, dass die Verwendung des Mappings für High-Performance-Zugriffe geeignet wäre, wird es in der vorliegenden Masterarbeit nur selten genutzt. Stattdessen werden klassische SQL-Abfragen zur Kommunikation verwendet, um sicher zu gehen, dass die bestmögliche Leistung erzielt werden kann. Ferner fiel die Entscheidung auf *SQL-Alchemy*, weil so eine bessere Möglichkeit besteht, bei Bedarf das Datenbankmangementsystem zu wechseln, ohne große Anpassung am Code vornehmen zu müssen.

## 4.2 Schnittstellen zur Zitationsdatenbank

### 4.2.1 Die Klasse *db_instance*

Die in Python implementierte Klasse *db_instance* stellt eine Basisklasse dar, die alle Grundfunktionalitäten bereitstellt, um beispielsweise die Verbindung zu einer beliebigen Datenbank herzustellen bzw. zu trennen, mithilfe eines gegebenem Skriptes Tabellenstrukturen zu erstellen und deren Indizes zu optimieren oder gänzlich von Daten zu befreien. Zudem ist die Klasse in der Lage, beliebige SQL-Kommandos an die Datenbank und auch deren Ergebnis weiterzuleiten. Das Klassendiagramm ist in Abb. 4.1 ersichtlich.

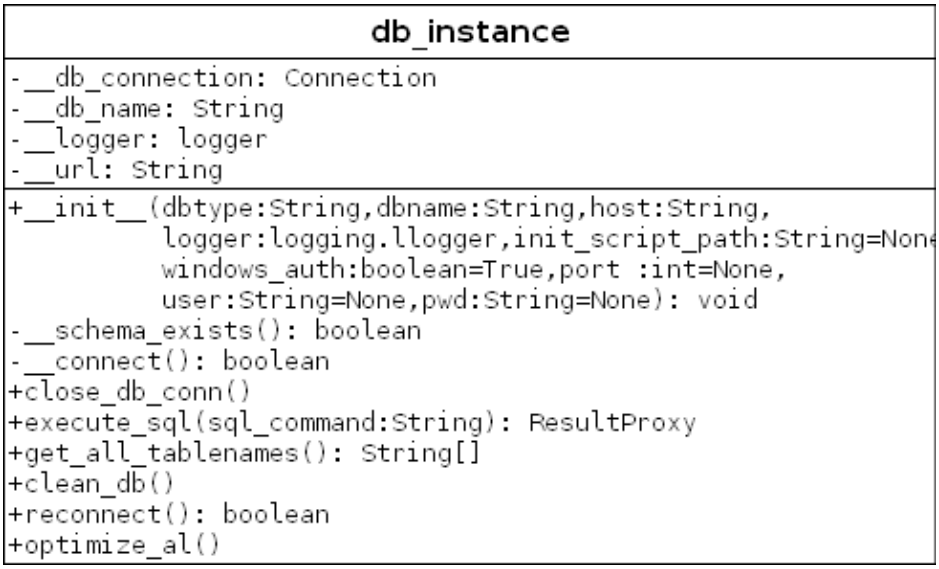



Abbildung 4.1: Klassendiagramm der Klasse db_instance

### 4.2.2 Die Klasse *citation_db*

#### Erstellung des Schemas

Die Klasse *citation_db* ist das Kernstück der Anwendung. Sie erweitert die universelle Klasse *db_instance* um die meisten Funktionalitäten, die zum Einladen, Aktualisieren und Nutzen der Zitationsdatenbank erforderlich sind. Eine Instanz der Klasse ist speziell auf das Schema der jeweiligen Zitationsdatenbank abgestimmt. Weil eine solche Datenbank für unterschiedliche Metadaten funktionieren muss, wird die Klasse über eine Konfigurationsdatei gesteuert. So kann erreicht werden, dass auch Bibliometriker ohne Datenbank- und Programmierkenntnisse in der Lage sind, das Schema der Datenbank richtig aufzusetzen. Das Format dieser Konfigurationsdatei kann der Beispieldatei in Abb. 4.3 entnommen werden. Informationen zu den einzelnen Parametern werden in Abbildung 4.2 dargestellt.

| Parameter | Beschreibung |
|---|---|
| **DB_TYPE** | Datenbankentyp. Genaue Bezeichnung ist in der Auflistung der Tabelle "Connect Strings" zu entnehmen. (siehe http://docs.sqlalchemy.org/en/rel_0_7/core/engines.html) |
| **HOST** | (vollqualifizierte) URL des Servers. |
| **SCHEMA** | Name des Schemas. |
| **PORT** | Port. |
| **USER** | Name des Datenbanknutzers, der Privilegien für Update, Insert, Select, Create, Drop, Index, References, Execute und Locks auf dem passenden Schema besitzt. |
| **PWD** | Unverschlüsseltes Passwort des Nutzers. (Verschlüsselung sollte in zukünftigen Versionen erfolgen!) |
| **WINDOWS_AUTH** | Authentifizierung ausschließlich durch aktives Windows-Benutzerkonto (0 für "Nein", 1 für "Ja"). Nötig bei MS-SQL Datenbanken. |
| **INIT_SCRIPT_PATH** | Absoluter Pfad, an dem das SQL-Skript zur Erzeugung des Schemas liegt. Das Python-Skript *create_init_script.py* nutzt den Parameter außerdem als Ausgabepfad. |
| **RECORD_TABLE_NAME** | Name der Tabelle, in der die Metadaten gespeichert werden sollen. |
| **REFERENCE_TABLE_NAME** | Name der Tabelle, in der die Referenzen der Metadaten gespeichert werden sollen. |
| **MAIN_ID** | Name der ID. Dieser muss unbedingt mit dem Namen der ID in den einzuladenden Metadaten übereinstimmen. |
| **MAIN_ID_TYPE** | Datentyp der ID. Erlaubt sind die folgenden Werte: *smallint*, *mediumtint*, *bigint*, *int*, *integer*, *char* bzw. den analog den von der Datenbank unterstützten Datentypen. |
| **MAIN_ID_SIZE** | Anzahl der Stellen/Zeichen der ID. |
| **CITE_ID** | Optional. Name der Zitat-ID. Muss unbedingt mit dem Namen der ID in den einzuladenden Metadaten übereinstimmen. |
| **CITE_ID_TYPE** | Nur nötig, wenn **CITE_ID** angegeben ist. **MAIN_ID_TYPE** entsprechend. |
| **CITE_ID_SIZE** | Nur nötig, wenn **CITE_ID** angegeben ist. Anzahl der Stellen/Zeichen der Zitat-ID. |

Abbildung 4.2: Auflistung aller Optionen der Konfigurationsdatei

Wurde die Konfigurationsdatei korrekt definiert, kann mithilfe des Python-Skriptes *create_init_script.py* ein SQL-Initialsierungsskript in Abhängigkeit der konfigurierten Werte erstellt werden. Das SQL-Skript funktioniert bisher nur für MySQL-Datenbanken und muss pro Konfigurationsdatei, also im Prinzip pro Datenbank, die man erstellen möchte, nur einmalig erstellt werden.
Die Klasse *citation_db* führt das SQL-Initialsierungskript bereits im Konstruktor aus, wodurch die Anwesenheit der erforderlichen Tabellen sichergestellt wird.

```
[Connection]

DB_TYPE=mysql
HOST=ZB0116
SCHEMA=scopus_citation_db
PORT=3306
USER=citation_db
PWD=streng_geheim!
WINDOWS_AUTH=0

[Schema-Info]

INIT_SCRIPT_PATH=/home/user_a/my_path/citation_db/scopus.conf
RECORD_TABLE_NAME=Record
REFERENCE_TABLE_NAME=refers_to

[ID-Info]

MAIN_ID=SGR
MAIN_ID_TYPE=bigint
MAIN_ID_SIZE=15
```

Abbildung 4.3: Beispiel für Konfigurationsdatei

### Einladen

Um die Metadaten in die Datenbank einzuladen, bietet die hier vorgestellte Klasse die Methode *insert_record(...)* an. Dabei wird ein Python-Dictionary als Übergabewert verlangt, dessen Schlüssel den Namen der Spalten und deren Werte den zugehörigen Inhalten entspricht. Zwar sind die meisten Spaltennamen, abgesehen von den IDs, durch das Schema und somit durch das teilweise hartcodierte Initialisierungsskript festgelegt, trotzdem ist so eine

Änderung in der Zukunft leichter zu realisieren. Ein beispielhaftes Python-Dictionary, das einen Datensatz repräsentiert ist im Folgenden dargestellt.

```
table_dict = \
{
SGR : 14835031679,
   'times_cited' : 0,
    'ref_count' : 14,
    'author_count' : 2,
    'pub_year' : 2001,
    'doc_type' :  "article",
    'journal' : "n.A.",
    'subject_cat' : "n.A",
    'DOI' : "10.1000/182"
  }
```

Neben dem Dictionary, dass alle Metadaten enthält, muss auch eine Liste mit allen IDs, die der einzufügende Datensatz referenziert, angegeben werden. Zudem kann eine Puffergröße, auf deren Bedeutung später eingegangen wird, eingestellt werden.

Die Nutzung der Methode als Schnittstelle ist leider sehr technisch. Es ist von Nöten, dass ein Skript erstellt wird, welches in der Lage ist, das vorliegende Datenformat zu lesen, damit ein Dictionary zu füllen und jenes dann an eine Instanz der Klasse *citation_db* zu übergeben. Sinnvoll wäre es, wenn auch das Einladen der Daten erledigt werden könnte, ohne dass eine Zeile Code geschrieben werden muss. So würde auch hier eine möglichst breite Masse an Benutzern angesprochen werden. Weil aber die Formate der Metadaten so stark variieren können (siehe hierzu Kapitel 2.2.2), ist es unmöglich, einen universellen Parser zu entwickeln, der jedes der Formate lesen und die Datenbank damit füllen kann. Zwar könnte man ein bestimmtes Zwischenformat wie beispielsweise JSON nutzen, sodass der Benutzer das Einladen mit einem einfachen Kommandozeilen-Aufruf tätigen kann, jedoch benötigt man dann noch immer einen Programmierer,

der die einzuladenden Daten zuerst in das Zwischenformat konvertiert. Da die Konvertierung bei der vorliegenden Masse an Daten sehr viel Zeit beansprucht, erscheint die vorliegende programmiersprachennahe Lösung als sinnvoller da zeitsparender.

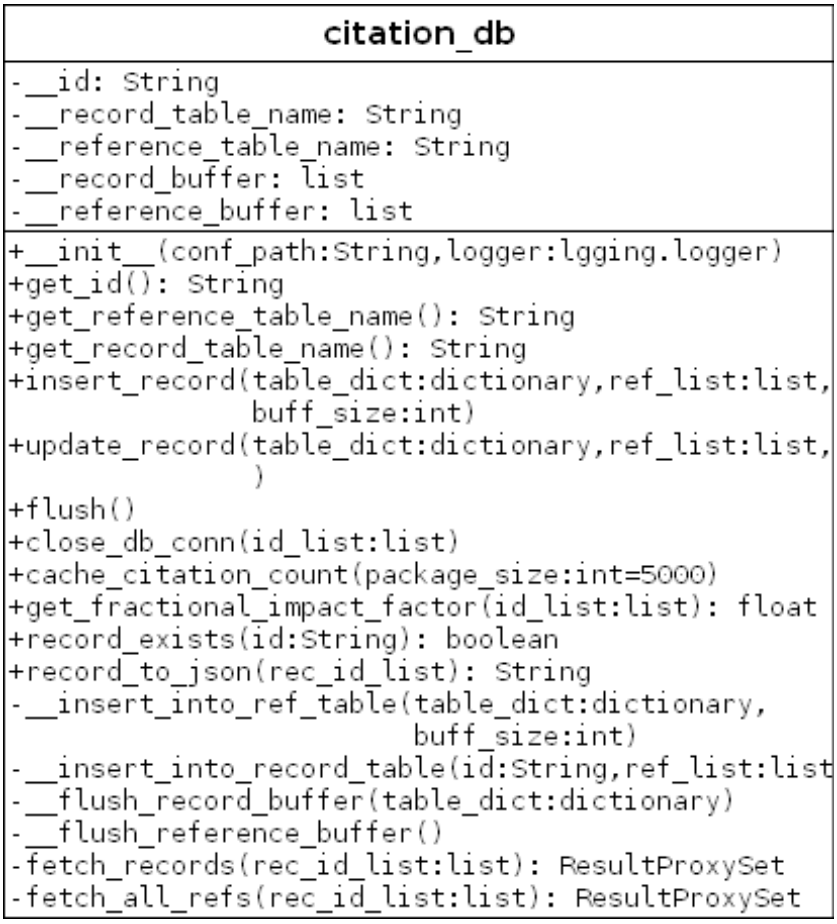



Abbildung 4.4: Klassendiagramm der Klasse *citation_db*

Im nächsten Abschnitt soll auf die Geschwindigkeit des Einladens eingegangen werden. Diese ist zum Einen davon abhängig, wie lange das Auslesen der Daten aus den Textdateien dauert. Je komplexer das Format, desto länger dauert der entsprechende Part des Einladens. Dem Format, in dem die Scopus-Metadaten geliefert wurden, liegt beispielsweise (wie bereits erwähnt) ein sehr umfangreiches XML-Schema zugrunde. Aufgrund der Komplexität

| **Optimierung** | **Vorteil** | **Nachteil** |
|---|---|---|
| *Index deakivieren* | weniger Overhead | Prüfung auf Eindeutigkeit geht verloren, Teil der Arbeit wird verschoben |
| *Puffer mit LOAD DATA LOCAL INFILE* | weniger Datentransfer, weniger Festplattenzugriffe | keine Kontrolle über doppelte Einträge |
| *Regelmäßiges OPTIMIZE* | schnelleres Einfügen in Indizes | zusätzlicher Overhead |

Tabelle 4.1: Überblick über Optimierungsmöglichkeiten von *INSERT*

nimmt das Parsen dieses XML-Schemas mit üblichen XML-Parsern sehr viel Zeit in Anspruch. Deshalb ist auch eine geringe Einladegeschwindigkeit zu erwarten.
Zum Anderen hängt die Gesamtgeschwindigkeit, mit der eingeladen wird, von datenbankseitigen Faktoren ab. Um sie so gut wie möglich zu steigern, gibt es einige Verbesserungen, die angewendet werden können.
Wie bereits in Abschnitt 3.3.4 erläutert, sind Indizes ein sehr großer Faktor, der negativen Einfluss auf die Geschwindigkeit des *INSERT*-Befehls nimmt. Beim Einfügen jeder Zeile muss unter Umständen auf Eindeutigkeit geprüft und die Indizes erweitert werden. Deshalb macht es Sinn, manche Indizes vorübergehend zu deaktivieren und erst nach vollständigem Einladen der Daten komplett berechnen zu lassen. Im Falle der vorhandenen Metadaten ist die Überprüfung der doppelten Einträge (Primärschlüssel und Referenzen-Paar) nötig, weshalb nur zwei von vier Indizes auf diese Art und Weise deaktiviert werden können, um einen Geschwindigkeitsvorteil zu erhalten.
Ein weiterer Optimierungsprozess stellt die Nutzung eines Puffers dar. Die

Instanz der Klasse *citation_db* erhält zur Laufzeit einzelne Datensätze. Statt jeden Datensatz einzeln via *INSERT*-Befehl an die Datenbank zu schicken, wird eine bestimmte Anzahl an Sätze im Arbeitsspeicher gesammelt. Ist die Puffergröße, steuerbar über den Parameter *buff_size*, erreicht, wird ein *LOAD DATA INFILE*-Kommando vorbereitet. Hierbei handelt es sich um eine Version des *INSERT*-Befehls, die eine ganze Textdatei voller Datensätzen an den Server schickt. Das Format dieser Textdatei ist frei definierbar, oftmals wird aber ein CSV-ähnliches Format genutzt. Ratsam ist es, das Einladen auf dem gleichem Server zu erledigen[1], damit die Textdatei nicht über das Netzwerk geschickt werden muss, wodurch ein weiterer Flaschenhals entstehen könnte. So kann ein deutlicher Leistungsgewinn verzeichnet werden[23], da MySQL den internen Index-Buffer erst leert, nachdem alle einzufügenden Zeilen aus der Insert-Datei ausgelesen und verarbeitet wurden. Dieser Vorgang hat jedoch auch einen Nachteil. Nutzt man den Befehl *LOAD DATA LOCAL INFILE*, also lokal auf dem Datenbankenserver, hat man keine Kontrolle mehr über die Reaktion auf doppelte Einträge. Enthält eine Datei einen Eintrag, der mit der Eindeutigkeits-Überprüfung der Indizes kollidiert, wird dieser Eintrag einfach ignoriert, ohne dass die Anwendung etwas davon mitbekommt. Auf diese Weise kann es passieren, dass Fehler, die beim Verarbeiten des Metadaten-Formates enstanden sind, unentdeckt bleiben. Dennoch wird das lokale Einladen in dieser Arbeit verwendet.
Abschließend bringt es einige Vorteile, wenn man den Befehl *OPTIMIZE TABLES* regelmäßig während des Einfügeprozesses aufruft. Durch ihn entfernt MySQL Fragmentierungen im Index, die vorallem durch Löschen und Updaten, aber auch durch Einfügen entstehen können. Fragmentierungen führen zu längeren Suchvorgängen im Index und somit zu einer längeren Einladezeit. Die *citation_db* bietet hierzu die geerbte Methode *optimize_all()* an, die man beispielsweise nach jeder Million eingeladenen Datensätzen aufrufen kann.
Tabelle 4.1 bietet einen Überblick über alle eben genannten Optimierungsoptionen.

[1]Um Netzwerktransfer über den *localhost* zu vermeiden, sollte man *LOAD DATA LOCAL INFILE* verwenden

### Vorberechnung der Zitationszahl

Der wichtigste Wert, der später aus der Datenbank abgefragt wird, ist die Zitationszahl der Datensätze. Bei größeren Abfragen kann es durchaus vorkommen, dass dieser Wert für mehrere Millionen Datensätze bestimmt werden muss. Um die Antwortzeit dieser Abfragen während der Laufzeit der Anwendung zu verbessern, ist es ratsam, die Zitationsrate aller vorhandenen Datensätze vorher zu bestimmen und zu speichern. Die Spalte *Record.times_cited*, die mit „0“ vorinitialisiert ist, ist für diesen Zweck vorgesehen. Um diese Zahl aus der Datenbank zu ermitteln, muss prinzipiell für jede ID, die in der Tabelle *Record* existiert, gezählt werden, wie oft sie in der Spalte *refers_to.cite_id* vorkommt. Ein Vorkommen in dieser Spalte bedeutet nämlich, dass sie ein anderer Datensatz als Referenz angibt, was wiederum in der Erhöhung der Zitationszahl resultiert. Eine beispielhafte Abfrage, welche die Zählung für jeden Datensatz übernimmt, sähe also so aus:

```
UPDATE times_cited
SELECT ref.ID, count(ref.cite_id) as times_cited
  FROM refers_to ref
  JOIN (Record) ON (Record.ID = ref.ID)
  GROUP BY ref.ID
```

Problematisch an dieser Vorgehensweise ist das *JOIN* auf zwei so riesige Tabellen. Zwar sind sowohl die Felder *ID* und *ref_ID* indiziert, sodass die *JOIN*-Bedingung und auch das *GROUP BY* schnell ist, aber das Zwischenergebnis, dass durch die Verknüpfung der beiden Tabellen entsteht, ist extrem groß. Zwar lässt sich die Abfrage mittels *LIMIT <start >, <menge >* portionieren, hierbei wird bei der verwendeten MySQL-Version scheinbar nur das Endergebnis eingeschränkt, nicht jedoch das Zwischenergebnis des *JOIN*. Deshalb muss diese Problematik auf Python-Ebene gelöst werden. Die Methode *cache_citation_count()* holt sich päckchenweise alle vorhandenen IDs aus der Tabelle *Record*, zählt deren Zitationsrate mittels *count()* in der Tabelle *refers_to* und schreibt den ermittelten Wert an die entsprechende Stelle in die Spalte *Record.times_cited*. Die Paketgröße ist über den Parameter *package_size* einstellbar. Leider bietet MySQL keine hochperformante

Möglichkeit, eine große Menge an Daten mittels *UPDATE* zu verändern, wie es beim *INSERT* mithilfe des *LOAD DATA LOCAL INFILE*-Kommandos möglich war. Dennoch lässt sich die Geschwindigkeit deutlich steigern, indem nicht einzelne *UPDATE*-Kommandos an die Datenbank schickt, sondern alle Befehle des gesamten Paketes angehäuft und diese dann gesammelt transferiert werden. Durch Verwendung des *LOCK TABLE Record WRITE*-Befehls vor dem Versenden eines Paketes, gefolgt vom Kommando *UNLOCK TABLES* am Ende eines jeden Paketes, wird der MySQL-Server veranlasst, seinen Index-Buffer erst nach der Transaktion zu entleeren. Diese Vorgehensweise spart viele Festplattenzugriffe und führt zu einem ähnlichem Effekt wie es bei *LOAD DATA LOCAL INFILE* der Fall ist.
Um die Zitationszahlen aktuell zu halten, muss darauf geachtet werden, dass die Methode *cache_citation_count()* unbedingt nach jeder Änderung an der Datenbank aufgerufen wird. Für den Benutzer steht hierfür das Skript *cache_tc.py* zur Verfügung, welches im Prinzip nur ein Wrapper darstellt, der die eben genannte Methode via Kommandozeile aufruft.

### 4.2.3 Ermittlung der Zitationszahl

Sind die Zitationszahlen in der Datenbank erst einmal gespeichert, kann man sich dieser Werte zur Laufzeit jederzeit schnell bedienen. Um nun zu einem gegebenen Satz von Datensätzen die gesamte Zitationszahl zu ermitteln, steht die Methode *get_citation_count(...)* bereit. Diese verlangt eine Liste von Datensätzen, repräsentiert durch ihre IDs, sowie eine Puffergröße als Parameter. Für jeden gegebenen Datensatz wird der Zitationswert in der Datenbank nachgeschlagen und zu einem Gesamtwert aufsummiert. Dieser wird dann als Ergebnis des Funktionsaufrufs geliefert.

#### Updaten

Manche bibliographische Sammlungen wie beispielsweise das *Web of Science* bieten zusätzlich zu regelmaßigen neuen Datensätzen auch Korrekturen zu bestehenden an. Sie beseitigen fehlerhafte Einträge, Werte oder Referenzen. So kann es von Sammlung zu Sammlung unterschiedlich sein, ob zu jedem

zu korrigierenden Datensatz nur die Änderung oder der ganze Datensatz angegeben ist. Die *citation_db*-Klasse bietet zu diesem Zwecke die Methode *update_record(...)* an. Die verlangten Parameter sind identisch zu denen, die bereits für die Methode *insert_record(...)* beschrieben wurden. Der einzige Unterschied ist hier jedoch, dass die Angabe einer Liste mit Referenzen optional ist, da diese möglicherweise nicht korrigiert werden müssen.
Im Wesentlichen werden im Ablauf des Prozesses paketisierte *UPDATE*-Befehle an die Datenbank gesendet, wie es schon im Kaptitel 4.2.2 beschrieben wurde. Ist eine Referenzliste angegeben, wird zuerst die alte Liste aller Referenzen abgerufen. Anschließend können mithilfe von Python-Operationen Änderungen bestimmt werden. Hierzu müssen die Listen zu einem Set konvertiert werden, um sich danach den (performanteren) Subtraktionen von Sets zu bedienen. Die genaue Vorgehensweise wird mithilfe des konkreten Quellcodes verdeutlicht:

```
#get all old references
old_list = set(self.fetch_all_refs(table_dict[self.__id]))
new_list = set(new_list)

to_add = new_list - old_list
to_remove = old_list - new_list
```

Diese Vorgehensweise beansprucht zwar mehrere Rechenoperationen auf Python-Ebene, spart aber im Average-Case einige *DELETE* bzw. *INSERT*-Befehle auf Datenbank-Ebene, welche üblicherweise deutlich langsamer als die erwähnten Rechenoperationen sind. Abschließend muss die Zahl der Referenzen, welche aus Perfomanzgründen redundant in der Datenbank gespeichert ist, aktualisiert werden.
Nach dem Update-Vorgang müssen auch die Zitationszahlen neu berechnet werden.

### 4.2.4 Ausgabe als JSON

Mithilfe der Methode *record_to_json(...)*, welche genauso eine Liste von IDs verlangt, ist es möglich, alle Informationen der in der Liste angegebenen Da-

tensätze im JSON-Format auszugeben. Auf diese Weise können die Informationen, die man aus der Zitationsdatenbank erhält, in weiteren Abläufen von anderen Anwendungen verarbeitet werden. Jeder Datensatz wird als JSON-Objekt, mit seinen Metadaten als Attribute inklusive Referenzen, dargestellt.

### 4.2.5 Ermittlung des Fractional Impact Factors

Nun soll die Ermittlung des Fractional Impact Factors betrachtet werden. Dieser ähnelt der Zitationsbestimmung prinzipiell sehr. Leider kann dennoch nicht auf die in der Datenbank gespeicherte Zitationszahl zurückgegriffen werden, da hier bereits die Anzahl aller zitierenden Datensätzen aufsummiert ist. Benötigt wird jedoch die Summe der durch die Anzahl der Referenzen der zitierenden Datensätze normierte Zitationszahl. Deshalb müssen diese Werte zur Laufzeit aus der Datenbank bestimmt werden.
Die Methode *get_fractional_impact_factor(...)* erwartet ebenfalls eine Liste von IDs. Zuerst werden alle IDs mithilfe der Methode *get_cited_by_list(...)*, ermittelt, welche die gegebene Datensätze zitieren. Für jede ID in dieser ermittelten Liste wird die Anzahl der Referenzen $r$ ermittelt. Zu diesem Zwecke liegt die redundante Anzahl in der Tabellen-Spalte *ref_count* bereit. Anschließend wird der Kehrwert $\frac{1}{r}$ zu einem Gesamtwert aufaddiert und am Ende durch die Zahl der Eingabe-IDs dividiert (für nähere Erläuterung vergleiche Kapitel 2.1.2).

### 4.2.6 Die Klasse *dual_id_citation_db*

Bei der Klasse *dual_id_citation_db* handelt es sich um einen Sonderfall. Sie ist von *citation_db* abgeleitet und erbt somit ihre Funktionen und Eigenschaften. Der Unterschied zwischen diesen beiden Klassen ist die Anzahl der IDs. Die Zitationsdatenbank, die durch die Klasse *citation_db* aufgebaut wird, ist für Daten, wie sie beispielsweise in Scopus enthalten sind, gedacht. Jeder Datensatz hat eine ID, über die man ihn eindeutig identifizieren kann. Auch die Angabe der Referenzen jedes Datensatzes erfolgt über diese ID.
Datenbanken, die hingegen mithilfe der *dual_id_citation_db*-Klasse aufgebaut

werden, haben, wie der Name bereits vermuten lässt, zwei entscheidende IDs. Zum Einen die, die jeden Datensatz eindeutig identifiziert, und zum Anderen eine, die bei der Referenzenliste benutzt wird (hier meist *cite_id* genannt). Prinzipiell ist es zwar möglich, die Zweite durch die Erste zu ersetzen, da jeder Datensatz, der von einem anderem referenziert wird, beide IDs enthält. Allerdings muss der korrespondierende Datensatz bereits in der Datenbank stehen, um die zweite ID aufzulösen, was während der Einladephase nicht garantiert werden kann. Deshalb werden beide Identifikatoren gespeichert und beim Zählen der Zitationsrate ad-hoc miteinander verknüpft. Das Klassendiagramm ist in Abbildung 4.5 skizziert. Es werden nur zusätzliche oder überschriebene Informationen aufgelistet. Alle anderen, von der Vaterklasse geerbten Methoden und Attribute, wurden aus Gründen der Übersichtlichkeit vernachlässigt

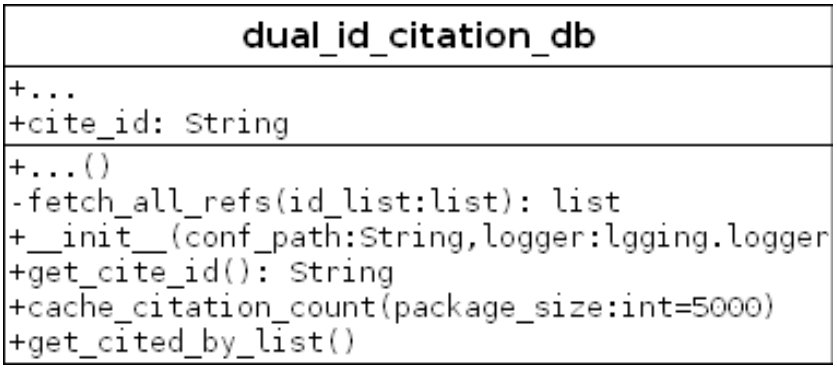



Abbildung 4.5: Klassendiagramm der Klasse *dual_id_citation_db*

Um das eben beschriebene Modell abzubilden, besitzt die Klasse *dual_id_citation_db* eine weitere Klassen-Variable, die den Namen der *cite_id* speichert. Zusätzlich wurden einige Methoden überschrieben, da sich die Bestimmung der Zitationen aufgrund oben genannter Tatsachen etwas von der Vaterklasse *citation_db* unterscheidet.

### 4.2.7 Die Klasse *citation_db_factory*

Diese sehr überschaubare Klasse besitzt eine einzige statische Methode *create_citation_db*, wobei es sich um eine klassische Fabrikmethode handelt. Sie erzeugt entweder eine Instanz von *citation_db* oder von *dual_id_citation_db*, abhängig davon, ob die Option *cite_id* in der Konfigurationsdatei gesetzt ist oder nicht. Die Konfigurationsdatei sowie eine Instanz eines Logger-Objektes, welches durch die Standard-Library *logging.logger* aus Python 2.7 definiert ist, werden als Parameter der Methode erwartet.
Die Verwendung dieser Fabrikmethode gewährleistet eine hohe Erweiterbarkeit, da die Anwendungen, in denen ein entsprechendes *citation_db*-Objekt genutzt wird, vom genauen Typ der Klasse gekapselt ist.

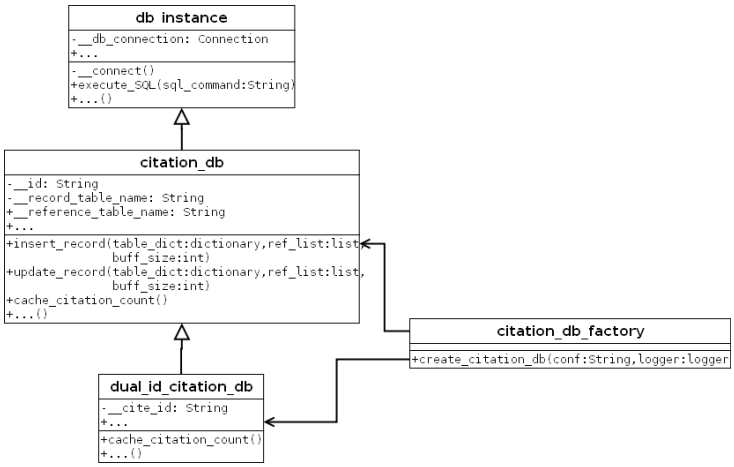



Abbildung 4.6: Überblick über alle Klassen

## 4.3 Schnittstelle zu den bibliographischen Daten

### 4.3.1 CDS Invenio

Die bibliographischen Datensätze werden hauptsächlich von Invenio verwaltet (siehe 3.1). Dort werden die kompletten Daten archiviert und via HTTP-Webservice durchsuchbar gemacht. Zudem bietet Invenio eine um-

fangreiche Suchsyntax, sodass die Bibliometriker in der Lage sind, komplexe Abfragen abzusetzen. So können beispielweise alle Datensätze angesprochen werden, die zwischen 2005 und 2007 in Deutschland veröffentlicht wurden, aus allen Fachgebieten außer der Biologie stammen und deren Autoren eine nicht-deutsche Nationalität aufweisen. Die einfache Anzeige im Browser via HTML ist die standardmäßige Ausgabeoption der Treffermenge. Sie kann auch in verschiedenen, beispielsweise für EndNote[25] oder Bibtex[26] spezifischen Formaten, ausgegeben werden. Zusätzlich können auch eigene Ausgabeformate definiert werden, die exzellent als Schnittstelle verwendet werden können. Ein Ausgabeformat, das schlicht und einfach die Identifikatoren aller Datensätze der Treffermenge als Plain-Text ausgibt bzw. in eine Datei schreibt, ermöglicht so die Kommunikation mit der Zitationsdatenbank.

### 4.3.2 *citation_counter.py*

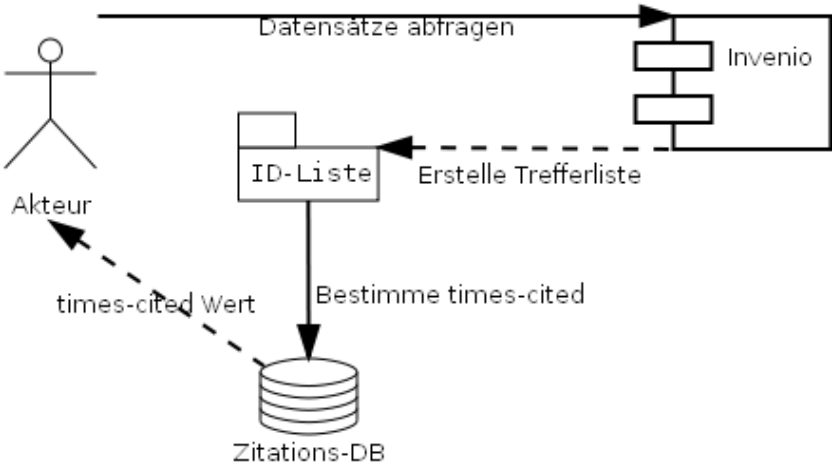



Abbildung 4.7: Ablauf der Zitationsbestimmung

Um die eigentlichen Features der Zitationsdatenbank nun via Kommandozeile zugänglich zu machen, steht das Skript *citation_counter.py* bereit. Als

Eingabe erwartet es eine Plain-Text-Datei, welche eine Liste von IDs beinhaltet. Das Abgrenzungszeichen, das die IDs voneinander abtrennt, ist frei einstellbar, wobei standardmäßig das Linefeed-Zeichen genutzt wird. Eine weitere erforderliche Angabe sind die Datenbankinformationen, die in Form einer Konfigurationsdatei, wie sie in 4.2.2 beschrieben ist, angegeben werden. Mithilfe dieser Konfigurationsdatei ist das Skript in der Lage, ein *citation_db*-Objekt anhand der *citation_db_factory* zu erstellen. Dieses Objekt kann nun mit der ID-Liste bedient werden. So besteht die Möglichkeit, zu allen durch ihre ID dargestellten Datensätze die aufsummierte Zitationszahl auszugeben, ihre zitierenden Datensätze im JSON-Format[24] zu liefern oder ihren Fractional Impact Factor zu ermitteln. Eine schematische Darstellung des Ablaufs im Gesamtsystem ist in Abbildung 4.7 ersichtlich.

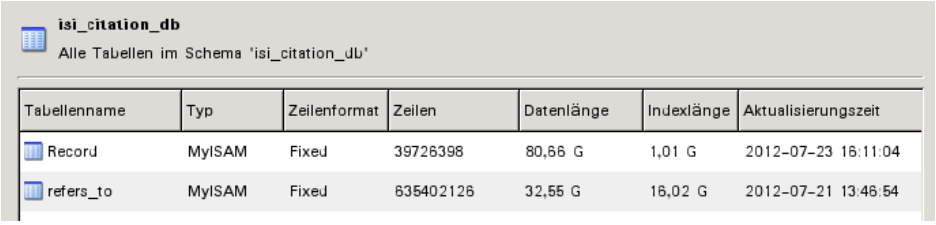
| **Parameter** | **Beschreibung** |
|---|---|
| **--conf=** | Path of the config-file. (required) |
| **--ids=** | Path of the file listing the IDs. (required) |
| **-o, --output** | Path of the output file. Only needed if --format=json or cited-by-json.<br>default: ./<name-of-id-file>.out |
| **--format=** | one of the following:<br><br>*tc-sum*: prints the total times-cited value of the given IDs (default)<br>*json*: writes the according records in a with -o specified file, formatted as JSON.<br>cited-by-json: writes the according citing records in a with -o specified file, formatted as JSON.<br>*fif*: prints the total fractional impact factor of the given IDs |
| **--log_path=** | Path of the log file. default: ./citation_counter.log |
| **--flush-size** | Since the amount of in- and output IDs is expected to be a large number, the queries are fragmented into many packages. This parameter controls the size of a single package. Normally, you don't need to specify this.<br>default: 5000 |
| **--delimiter** | Specifies the delimiter, which is used to separate the IDs in the given input file.<br>Example content of an ID file with delimiter '\n' and 3 IDs:<br><br>123<br>456<br>789<br><br>If you use delimiter like ';' (example: 123;456;789), please take care of whitespace character like newline. default: '\n' |
| **-h, --help** | Prints this help. |

Abbildung 4.8: Beschreibung aller Parameter des Skriptes citation_counter.py

## 4.4 Analyse der Performanz

### 4.4.1 Vorwort

In diesem Kapitel soll die Performanz der im Rahmen dieser Arbeit entwickelten Anwendung untersucht werden. Es liegen, wie anfangs erläutert, zwei konkrete Datenquellen (Scopus, Web of Sciences/ISI) vor. Die Zitationsdatenbank bzw. der Programmcode zum Einladen und Berechnen ist zwar darauf ausgelegt, beliebige Datenquellen zu verwenden, die Analysen dieses Abschnittes beschränken sich aus Zeitgründen allerdings auf die Verwendung der ISI-Daten. In Abbildung 4.9 ist der Datenumfang ersichtlich.



**isi_citation_db**

Alle Tabellen im Schema 'isi_citation_db'

| Tabellenname | Typ | Zeilenformat | Zeilen | Datenlänge | Indexlänge | Aktualisierungszeit |
|---|---|---|---|---|---|---|
| Record | MyISAM | Fixed | 39726398 | 80,66 G | 1,01 G | 2012-07-23 16:11:04 |
| refers_to | MyISAM | Fixed | 635402126 | 32,55 G | 16,02 G | 2012-07-21 13:46:54 |

Abbildung 4.9: Eckdaten der konkreten ISI-Citation-DB

Sie beinhaltet 39,726,398 Datensätze und 635,402,126 Referenzen. Jeder Datensatz hat also durchschnittlich fast 16 Referenzen. Der eindeutige Indentifikator (in vorherigen Kapiteln allgemein: *ID* genannt) jedes Datensatzes ist der sogenannte „UT-Code“, wobei es sich um eine Zeichenkette, bestehend aus 15 Zeichen (0-9, A-Z), handelt. Referenzen werden über einen zweiten Identifikator, die sogannte „ItemIdNumber“ (in vorherigen Kapiteln allgemein: *cite_id* genannt), angegeben. Diese wird durch eine zehnstellige Zahl dargstellt. Nicht jeder Datensatz besitzt eine *ItemIdNumber* und nur ca. 22 Millionen aller in den Referenzen enthaltenen *ItemIdNumber* können zu einem UT-Code aufgelöst werden. Dies liegt daran, dass das Web of Science nicht alle Artikel, die es gibt, abdeckt.

Dadurch, dass zwei Identifikatoren enthalten sind, die bei einer Zitationsberechnung zueinander aufgelöst werden müssen (siehe auch Kapitel

4.2.6), entsteht ein gewisser Mehraufwand, der sich verglichen mit Zitationsdatenbanken anderer Datenquellen negativ auf die Performanz auswirken wird. Auch die Tatsache, dass der UT-Code eine Zeichenkette ist, ist von Nachteil, da sich Buchstaben üblicherweise schlechter untereinander vergleichen lassen, als Zahlen. So wird das Aufbauen und Nutzen des B-Baum-Indizes (und somit die gesamte Laufzeit) verhältnismäßig länger dauern.

### 4.4.2 Einladen und Updaten

Da ständig viele neue Publikationen verfasst werden, wächst üblicherweise auch die Menge der Datensätze, die eine bibliographische Sammlung beinhaltet, relativ schnell. So werden beispielsweise jährlich neue Scopus-Datensätze geliefert. ISI bietet neue Datensätze bei Bedarf sogar monatlich zum Download an[2]. Um aktuelle bibliogmetrische Analysen durchzuführen, ist in vielen Fällen auch der aktuellste Datenbestand erforderlich. Zwar ist das Einladen, verglichen mit den Berechnungen der Metriken, am wenigsten zeitkritisch, da dies nur einmalig geschehen muss. Dennoch muss die benötigte Zeit so gering gehalten werden, dass neue Daten eingeladen und anschließend im Rahmen mehrerer Analysen genutzt werden können, bevor aktuellere Datensätze veröffentlicht werden.
Wie bereits erläutert, konnten während des Einladevorgangs nicht alle Indizes deaktiviert werden. Je größer ein Index wird, desto länger dauert es, in ihn einzufügen.

[2]Natürlich nur, wenn die Sammlung vorher erworben wurde.

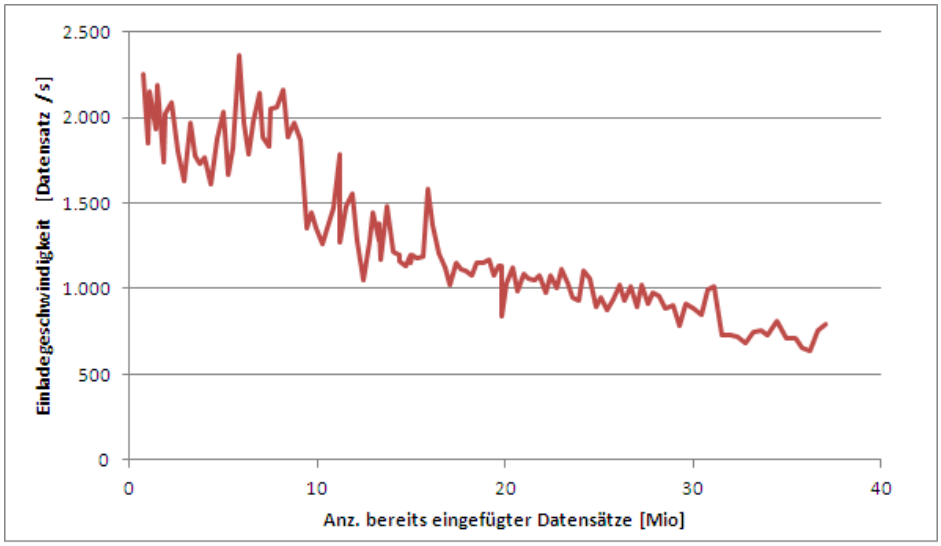



Abbildung 4.10: Übersicht über die Einladegeschwindigkeit

Dieser Effekt ist in Abbildung 4.10 deutlich erkennbar. In diesem Diagramm ist die Einladegeschwindigkeit in Abhängigkeit der bereits eingefügten Datensätze aufgetragen. Je mehr Daten eingeladen wurden, desto mehr nimmt die Geschwindigkeit ab. Die gesamte Geschwindigkeit ermittelt sich hierbei aus zwei Werten. Zum einen durch die konstante[3] Geschwindigkeit, mit dem die SQL-Befehle interpretiert und die Daten zur Datenbankengine übertragen werden[4] und zum anderen die Geschwindigkeit, mit der der Index erweitert wird. Bei letzterem handelt es sich um einen Prozess, der sich logarithmisch zur Größe des Indexes verhält.
Diese Daten beziehen sich lediglich auf das Einfügen der Referenzen. Informationen des kompletten Einladeprozesses liegen nicht vor.

### 4.4.3 Vorberechnung der Zitationszahl

Da beim Bestimmen der Zitationszahl ebenfalls auf den Index zugegriffen wird, ergibt sich die Geschwindigkeit dieses Prozesses aus einer Kombina-

[3]Zuzüglich einer Zufallsvariable mit unbekannter Verteilung, die vom OS ausgeführe nebenläufige Prozesse mit hoher Priorität beschreibt.

[4]In diesem Fall handelt es sich aufgrund der Nutzung von *LOAD DATA LOCAL INFILE* nur um Festplattenzugriffe statt Netzwerkverkehr.

tion mehreren logarithmischen und einigen konstanten (beispielsweise das *UPDATE* der neuen Zitationszahl) Geschwindigkeiten. Für die konkrete ISI-Zitationsdatenbank unter Verwendung einer Abfragepaketgröße von 50,000 ergibt sich hier eine Durchschnittsgeschwindigkeit von 399 Datensätze pro Sekunde, zu denen die Zitationszahl bestimmt und der entsprechende Wert in der Datenbank gespeichert werden kann. Dies führt zu einer gesamten Laufzeit von 27 Stunden, was ein akzeptables Ergebnis ist, weil dieser Prozess nur ausgeführt werden muss, wenn neue Daten eingespielt werden. Die Geschwindigkeit ist deshalb als Durchschnitt angegeben, da sie sich mit fortschreitender Laufzeit verschlechtert. Wie in Kapitel 4.2.2 beschrieben, wird die Zitationszahl paketweise abgerufen. Die Paketisierung geschieht mithilfe von *LIMIT* $<n>$, $<menge>$. MySQL ist leider nicht in der Lage, direkt zur *n-ten* Zeile im Index zu springen, sondern muss zuerst n Werte aus dem Index lesen. Je höher dieser Offset ist, desto länger dauert die Abfrage eines Paketes.

### 4.4.4 Ermittlung der Zitationszahl

Im Folgenden soll die Laufzeit der Zitationsermittlung am konkreten Beispiel der ISI-Zitationsdatenbank bestimmt werden. Mithilfe des Python-Skriptes *citation_counter.py*, aufgerufen mit dem Parameter *format=tc-sum*, wird die gesamte Zitationszahl aller Datensätze, die anhand einer Liste ihrer UT-Codes spezifiziert werden, ermittelt. Die Werte entsprechen dem Durchschnitt von jeweils 10 Messungen. In Abbildung 4.11 sind die Messergebnisse grafisch dargestellt. Tabelle 4.2 erlaubt detailliertere Einblicke in die exakten Messwerte. Die Berechnungszeit steigt linear mit der Menge an Datensätzen, deren Zitationszahl berechnet werden sollen, an. Da die einzelnen Zitationszahlen bereits in der Tabelle gespeichert sind, handelt es sich lediglich um $n$ *SELECT*-Anweisungen, wenn $n$ Datensätze abgefragt werden. Zwar wird bei jeder Anfrage der Index verwendet, welcher eigentlich ein logarithmisches Verhalten aufweist, jedoch skaliert dieses Verhalten mit der Größe $m$ des Indexes und nicht mit der Menge der Abfragen.

| Puffergröße | Anz. IDs | benötigte Zeit [Sekunden] |
|---|---|---|
| 100 | 1,000 | 0.26 |
| | 10,000 | 0.24 |
| | 100,000 | 23.6 |
| | 1,000,000 | 244.94 |
| 1000 | 1,000 | 0.22 |
| | 10,000 | 2.16 |
| | 100,000 | 26.19 |
| | 1,000,000 | 254.64 |
| 10000 | 10,000 | 2.59 |
| | 100,000 | 24.40 |
| | 1,000,000 | 249.3 |

Tabelle 4.2: Zeitmessung der Zitationszahlenbestimmung

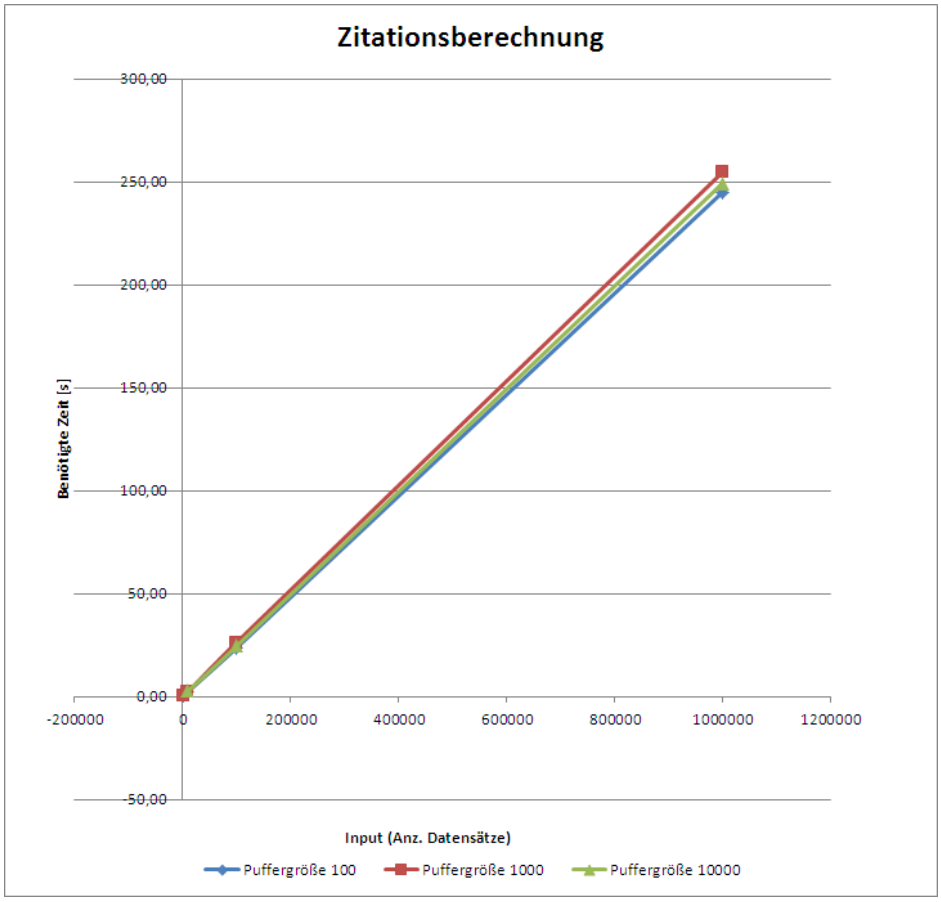



Abbildung 4.11: Übersicht über die Geschwindigkeit der Zitationsberechnung

| Anz. IDs | benötigte Zeit in s |
|---|---|
| 1,000 | 29 |
| 5,000 | 117 |
| 10,000 | 214 |
| 20,000 | 498 |
| 50,000 | 832 |
| 100,000 | 1867 |

Tabelle 4.3: Zeitmessung der Bestimmung des Fractional Impact Factors

Zudem scheint die Puffergröße keine relevante Auswirkung auf die benötigte Zeit zu haben. Auch diese Tatsache ist nicht besonders überraschend. Aus dem gesammelten Senden von $n$ *SELECT*-Abfragen kann keinerlei Vorteil gezogen werden, da nur $n$ lesende Zugriffe benötigt werden und keine Einsparung von Schreibvorgängen vorliegt, wie es bei *UPDATE* oder *LOAD DATA LOCAL INFILE* (siehe Kapitel 4.2.2) der Fall war. Betrachtet man die kleinen Zeitunterschiede der Puffergrößen spekulativ, erkennt man, dass die kleinste Puffergröße, nämlich 100, die schlechtesten Ergebnisse erzielt. Eine Puffergröße von 1,000 scheint hingegen optimal, die größte Puffergröße von 10,000 etwas schlechter. Es lässt sich vermuten, dass das Netzwerk die mittlere Puffergröße am besten handhaben kann. Um fundierte Aussagen diesbezüglich zu treffen, benötigt man jedoch eine viel umfangreichere Menge an Messdaten.

### 4.4.5 Ermittlung des Fractional Impact Factors

Nun soll die Ermittlung des Fractional Impact Factors betrachtet werden. Im Folgenden sind die Zeitmessungen, ermittelt aus dem Durchschnitt von 5 Durchläufen mit einer konstanten Puffergröße von 1000, aufgelistet.

Verglichen mit den Laufzeiten der Zitationsbestimmung liegen hier

deutliche höhere Zeiten vor, was an dem im Kapitel 4.2.5 beschriebenen Mehraufwand liegt. Für jeden UT-Code, der als Input angegeben ist, muss die entsprechende ItemIdNumber nachgeschlagen werden. Danach müssen alle UT-Codes bestimmt werden, welche die entsprechende ItemIdNumber zitieren. Dies kann MySQL direkt anhand des Indexes *(cite_id, ID)* der Tabelle *refers_to* ermitteln (siehe Kapitel 3.3.4). Anschließend muss ein letzter Zugriff auf die Tabelle *Record* erfolgen, um die Anzahl der Referenzen abzufragen. So benötigt man hier zwei Zugriffe auf die Tabelle *Record* und ein Zugriff auf einen Index.
Zudem ist die Laufzeit jetzt nicht nur abhängig davon, wieviele Datensätze abgefragt werden, sondern auch welche. Besonders bei einer kleinen Anzahl an Eingabe-IDs variiert die Laufzeit stark, da unter Umständen Datensätze mit besonders vielen oder wenigen zitierenden Datensätze abgefragt werden können.
Würde jeder Datensatz von $m$ anderen zitiert und wir wollen den Fractional Impact Factor zu $n$ Datensätzen bestimmen, werden insgesamt $n * m$ Tabellenabfragen an die Tabelle *Record* benötigt. Die Messergebnisse sind in Tabelle 4.3 zu sehen.

# Kapitel 5

# Bewertung

## 5.1 Zusammenfassung

Im Rahmen dieser Masterarbeit wurde eine Datenbank entworfen, die darauf ausgelegt ist, bibliometrische Massendaten zu speichern und auf Basis dieser Daten Zitationsanalysen durchzuführen. So kann zu einer Liste von Identifikatoren, die einen Artikel eindeutig identifizieren, bestimmt werden, welche andere Artikel ihn zitieren. Dies ist die Grundlage für bibliometrische Analysen und Zitationsmetriken wie den sogenannten Impact Factor.
Der Schwerpunkt des Entwurfs liegt auf einer möglichst hohen Performanz. So wurden einige mögliche server- und anwendungsseitge Optimierungen überprüft, analysiert und durchgeführt. Hierbei mussten manche Design-Standards wie Normalform und Transaktionssicherheit zugunsten der Laufzeit missachtet werden.
Zusätzlich wurden Funktionen bereitgestellt, die anhand von Python Skripten Zugriff auf die Datenbank erlauben. Diese wurden Anhand einer konkreten bibliographischen Sammlung, dem Web of Science, verwendet.

## 5.2 Fazit

Sowohl die Laufzeit vom Einladen der Daten als auch die Zählung des Zitationswertes ist hinreichend klein. Der gesamte Zitationswert von 1 Mio

Datensätze kann in weniger als 5 Minuten berechnet werden. Da letzteres die Hauptanwendung darstellt, ist das Ergebnis des Projektes gut. Die Bibliometriker vor Ort können die meisten Abfragen interaktiv tätigen, da sehr geringe Antwortzeiten vorliegen. Lediglich Abfragen, die mehrere Millionen Artikel als Ergebnis liefern, erfordern eine mäßige Wartezeit, welche den Arbeitsfluss aber kaum stören.
Aufgrund von sehr hohen Laufzeiten von Tests, Konfigurationen und Analysen, welche bei der Entwicklung nötig waren, sowie einer sehr hohen Komplexität des Optimierungsvorganges von MySQL, war dieses Projekt eine extrem (rechen-)zeitintensive Aufgabe. Das Ergebnis der Arbeit eignet sich sehr gut als Grundlage für eine fortschreitende Optimierung und Erweiterung um zusätzliche Features.

## 5.3 Ausblick

Wie schon im Fazit erwähnt, konnten nicht alle denkbaren bzw. wünschenswerten Features und Feinheiten umgesetzt werden, da diese den Zeitrahmen einer Masterarbeit sprengen.
Eine sehr praktischer Erweiterung würde eine Benutzeroberfläche darstellen, welche die wissenschaftler sowohl beim Einladen der Daten aber vor allem auch beim Durchführen der Analysen visuell unterstützt.
Weiterhin könnte man die Berechnung von weiteren, komplexeren Zitationsmetriken unterstützen. Zwar bietet die Zitationsdatenbank eine Schnittstelle in Form von JSON, die andere Anwendungen aufgreifen können, jedoch ist dieser Zwischenschritt weniger performant als Berechnungen direkt auf der Datenbank. Eine solche noch nicht implementierte, aber in dieser Arbeit erwähnte Metrik stellt beispielsweise der Impact Factor dar. Hierbei kämen Abfragen, die auf Jahr und Journal einschränken, zu den Use-Cases hinzu. Dies führt dazu, dass zusätzliche Indizes oder Partitionierungen nach diesen Feldern vorgenommen werden müssten.
Zudem sind weitere Optimierungen denkbar. Beispielsweise könnte man eine neuere MySQL-Version testen, die gegebenenfalls weitere Möglichkeiten

zur Optimierung wie Partitionierung bietet. Da die Anwendung speziell auf MySQL ausgelegt ist und sehr viele MySQL-spezifische Konfigurationen untersucht wurden, wäre es in Hinsicht der Konfiguration ein größeres Vorhaben, das Datenbankmanagementsystem gegen beispielsweise Oracle oder DB2 auszutauschen. Dennoch könnte eine solche Alternative ein besseres Performanzverhalten für die Anwendung aufweisen.
Ein zusätzlicher Ansatz stellt Parallelisierung dar. Die Datenbank könnte auf verschiedene Festplatten oder gar Cluster verteilt werden, sodass die Berechnungen parallel und somit schneller durchführbar sind.
Das Problem der zwei unterschiedlichen Identifikatoren, die bei manchen Sammlungen wie dem Web of Science auftauchen, könnte so verbessert werden, dass die Auflösung der IDs zueinander nicht jedes mal zur Laufzeit, sondern einmalig geschieht und dann gespeichert wird. Dadurch könnte die Laufzeit noch weiter verbessert werden.

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Anhang A

# Begriffserklärung

| | |
|---|---|
| **CDS Invenio** | Open-Source Software zur Archivierung von Publikationen. Siehe [15]. |
| **SGR-Nummer** | Eindeutiger Identifikator eines Scopus-Datensatzes. |
| **Read-Arround-Write** | Schreibvorgang des Betriebssystems, der nicht ohne einen vorherigen Lesezugriff auf die Festplatte durchgeführt werden kann. |
| **UT-Code** | Eindeutiger Identifikator eines Web of Science-Datensatzes. |
| **JSON** | JavaScript Object Notation. Ein kompaktes und leicht lesbares Format zum Austausch von Objekt-Informationen innerhalb von Anwendungen. Siehe [24]. |
| **XML** | Extensible Markup Language. Auszeichnungssprache zur Darstellung hierarchischer Strukturen. |

| | |
|---|---|
| **MARC** | Machine-Readable Catalog. Bibliothekarisches Format zur Speicherung der Metadaten von Literatur. |
| **Indexer** | Anwendung, welche die Aufbereitung von Daten zum schnellen Zugriff auf diese aufbereitet. |
| **Metadaten** | Daten, die Informationen über andere Daten beschreiben. Beispiel: Autor, Titel, Veröffentlichungsjahr eines Buches. |
| **Monitoring-Software** | Anwendung, die eine andere Anwendung bzw. ein System überwacht und Informationen über aktuelle Parameter des Systems anzeigt. |
| **Cache** | Zwischenspeicher eines Computers. Schnell aber teuer und deshalb Verhältnismäßig klein. Soll Flaschenhals zwischen schnellem CPU und langsamen Ein-/Ausgabe Operationen überbrücken. |
| **Swappen** | Vorgang des Betriebsystems, um Daten vom Datenträger in den Cache zu laden. |
| **Use-Case** | Anwendungsszenario, das auftreten kann, wenn ein Akteur mithilfe eines Systems ein bestimmtes Ziel erreichen will. |
| **O-Notation** | Landau-Notations. Beschreibt Laufzeit asymptotisch in Abhängigkeit der Eingabedatenmenge. |
| **transaktionssicher** | Einhaltung von Sicherheitsmaßnahmen bei der Durchführung von Transaktionen. (Atomarität, Konsistenz, Isolation, Dauerhaftigkeit) |

| | |
|---|---|
| **Raid 0** | Auch: Striping. Verbund von mehreren Festplatten, gesteuert durch einen Raid-Controller. Raid 0 verteilt Daten auf mehrere Festplatten, um parallel auf sie zugreifen zu können. Dies steigert die Ein-/Ausgabegeschwindigkeit des Systems. |

# Literaturverzeichnis


[1] Gorraiz, J. *„Bibliometrie“*, http://www.zbp.univie.ac.at/gj/citation/bibliometrie.htm, Stand: 02.05.2012

[2] Stock, W.G., Weber, S. (2006). *„Facets of Informetrics. Information. Wissenschaft und Praxis“*

[3] Nacke, O. (1979). *„Informetrie: Ein neuer Name für eine neue Disziplin“*

[4] Gorraiz, J. (1992). *„Die unerträgliche Bedeutung der Zitate“*

[5] Garfield, E. (1972). *„Citation analysis as a tool in research evaluation. Journals can be ranked by frequency and impact of citations for science policy studies“* Science, 178(4060), 471-479.

[6] Moed, H. F. (2005). *„Citation Analysis in Research Evaluation“*

[7] Glänzel,W., Schubert, A., Thijs, B., & Debackere, K. (2011). *„A priori vs. a posteriori normalisation of citation indicators. The case of journal ranking. Scientometrics“*, 87(2), 415424. doi: 10.1007/s11192-011-0345-6

[8] Leydesdorff, L. , & Opthof, T. (2010). *„Scopus Source Normalized Impact per Paper (SNIP) versus the Journal Impact Factor based on fractional counting of citations, Journal of the American Society for Information Science & Technology“*

[9] Leydesdorff, L., & Opthof, T. (2010b). *„Normalization at the field level:*

*Fractional counting of citations. Journal of Informetrics*“, 4(4), 644646. doi:10.1016/j.joi.2010.05.003

[10] Haustein, S. (2012). „*Multidimensional Journal Evaluation. Analyzing Scientific Periodicals beyond the Impact Factor. Berlin/Boston: De Gruyter Saur.*“

[11] Thomson Reuters - Web of Science *http://thomsonreuters.com/products_services/science/science_products/a-z/web_of_science/*, Stand: 02.07.2012

[12] SciVerse Scopus *http://www.info.sciverse.com/scopus*, Stand: 03.07.2012

[13] Library of Congress „*MARC Standards*“, http://www.loc.gov/marc/ Stand: 13.07.2012

[14] SQL-Alchemy Homepage *http://www.sqlalchemy.org/*, Stand: 02.07.2012

[15] CERN *http://invenio-software.org/wiki/General/Features*, Stand: 30.05.2012

[16] MONyog *http://www.webyog.com/en/*, Stand: 17.07.2012

[17] Schwartz B. Zaitsev, P. (2009) „*High Performance MySQL*“ 2.Auflage

[18] Tunger D., Ball R. (2005). „*Bibliometrische Analysen - Daten, Fakten und Methoden - Grundwissen Bibliometrie für Wissenschaftler, Wissenschaftsmanager, Forschungseinrichtungen und Hochschulen*“

[19] Weigend, M. (2006). „*Pyton - GE-PACKT*“ 3.Auflage

[20] Burnham, J. F. „*Scopus database: a review*“, *http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1420322/?tool=pmcentrez*, Stand: 14.03.2012

[21] MySQL Dokumentation: Speicher-Engines und Tabellentypen *http://dev.mysql.com/doc/refman/5.1/de/storage-engines.html*, Stand: 27.06.2012

[22] Liste von SQL-Alchemy Unterstützen Datenbanken *http://docs.sqlalchemy.org/en/rel_0_7/core/engines.html*, Stand: 02.07.2012

[23] Speed of INSERT Statements, MySQL-Guide *http://dev.mysql.com/doc/refman/5.0/en/insert-speed.html*, Stand: 02.07.2012

[24] Offiz. JSON-Webseite *http://www.json.org/*, Stand: 02.07.2012

[25] Offiz. EndNote-Webseite *http://www.endnote.com/*, Stand: 31.07.2012

[26] Offiz. BibteX-Webseite *http://www.bibtex.org/de/*, Stand: 31.07.2012

[27] Bar-Ilan, J. (2008). *„Informetrics at the beginning of the 21st century - A review“*, Journal of Informetrics 2, 1-52