



# **FernUniversität in Hagen**

– Fakultät für Mathematik und Informatik –  
Lehrgebiet Datenbanken und Informationssysteme

## **Multi-Layer Optimization Strategies for Enhanced Performance in Digital Editions: A Study on Database Queries, Caches, Java EE and JSF**

Abschlussarbeit zur Erlangung des akademischen Grades  
Bachelor of Science (B.Sc.)

vorgelegt von

**Marco Galster**

Matrikelnummer: 8335710

Referentin : Prof. Dr. Uta Störl

Betreuer : Sebastian Bruchhaus, Tobias Holstein

Marco Galster: *Multi-Layer Optimization Strategies for Enhanced Performance in Digital Editions: A Study on Database Queries, Caches, Java EE and JSF*, © 01.  
Januar 2024

## ERKLÄRUNG

---

Ich erkläre, dass ich die Bachelorarbeit selbstständig und ohne unzulässige Inanspruchnahme Dritter verfasst habe.

Ich habe dabei nur die angegebenen Quellen und Hilfsmittel verwendet und die aus diesen wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht.

Die Versicherung selbstständiger Arbeit gilt auch für enthaltene Zeichnungen, Skizzen oder graphische Darstellungen.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder derselben noch einer anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Mit der Abgabe der elektronischen Fassung der endgültigen Version der Arbeit nehme ich zur Kenntnis, dass diese mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate geprüft werden kann und ausschließlich für Prüfungszwecke gespeichert wird.

*Höchstadt, 01. Januar 2024*

---

Marco Galster

## ABSTRACT

---

*TODO: Dies am Ende noch Ausfüllen!!!*

A short summary of the contents in English of about one page. The following points should be addressed in particular:

- **Motivation:** Why did this work come about? Why is the topic of the work interesting (for the general public)? The motivation should be abstracted as far as possible from the specific tasks that may be given by a company.
- **Content:** What is the content of this thesis? What exactly is covered in the thesis? The methodology and working method should be briefly discussed here.
- **Results:** What are the results of this work? A brief overview of the most important results as a teaser to read the complete thesis.

BTW: A great guide by Kent Beck how to write good abstracts can be found here:

<https://plg.uwaterloo.ca/~migod/research/beck00PSLA.html>

## ZUSAMMENFASSUNG

---

*TODO: Dies am Ende noch Ausfüllen!!!*

Kurze Zusammenfassung des Inhaltes in deutscher Sprache von ca. einer Seite Länge. Dabei sollte vor allem auf die folgenden Punkte eingegangen werden:

- Motivation: Wieso ist diese Arbeit entstanden? Warum ist das Thema der Arbeit (für die Allgemeinheit) interessant? Dabei sollte die Motivation von der konkreten Aufgabenstellung, z.B. durch eine Firma, weitestgehend abstrahiert werden.
- Inhalt: Was ist Inhalt der Arbeit? Was genau wird in der Arbeit behandelt? Hier sollte kurz auf Methodik und Arbeitsweise eingegangen werden.
- Ergebnisse: Was sind die Ergebnisse der Arbeit? Ein kurzer Überblick über die wichtigsten Ergebnisse als Teaser, um die Arbeit vollständig zu lesen.

Eine großartige Anleitung von Kent Beck, wie man gute Abstracts schreibt, finden Sie hier:

<https://plg.uwaterloo.ca/~migod/research/beck00PSLA.html>

## INHALTSVERZEICHNIS

---

1	Einleitung	1
1.1	Ausgangslage	1
1.2	Ziel der Arbeit	2
1.3	Gliederung	2
2	Grundlagen	3
2.1	Glassfish - Enterprise Kirsten.Ritzmann@gmx.netJava Beans	3
2.2	Glassfish - Java Persistence API	4
2.3	Glassfish - OpenJPA Cache	5
2.4	PostgreSQL - Memory Buffers	5
2.5	PostgreSQL - Services	6
2.6	PostgreSQL - Abfragen	6
3	Konzept	8
3.1	Allgemeine Betrachtung des Systems	8
3.2	Untersuchung der Anwendung	9
4	Performance-Untersuchung	13
4.1	Auswertung des Systems	13
4.2	Statistiken im PostgreSQL auswerten	13
4.3	Überprüfung des PostgreSQL und Servers	13
4.4	Einbau und Aktivieren von Performance-Messung	13
4.5	Untersuchung der Anwendung	13
4.5.1	Caching im OpenJPA	14
4.5.2	Caching im Java Persistence API (JPA)	14
4.5.3	Caching in Enterprise Java Beans (EJB)	14
4.5.4	Abfragen JPQL	14
4.5.5	Abfragen Criteria API	14
4.5.6	materialized views	14
4.5.7	cached queries	14
4.5.8	Umgestalten der Datenbanktabellen	14
4.5.9	Verkleinerung der Abfragen	14
5	Optimierung	15
5.1	Ermittlung der Performance-Probleme	15
5.2	Analyse der Abfrage	15
5.3	Optimierungen der Abfragen	15
5.4	Anpassung der Konfiguration	15
6	Evaluierung	16
6.1	Erneute Laufzeitanalyse starten	16
6.2	Statistiken im PostgreSQL auswerten	16
6.3	Vergleich der Ergebnisse vor und nach der Optimierung	16
7	Zusammenfassung und Ausblick	17
I	Appendix	
A	Zeitmessung der Webseite	19

Literatur

21

## ABBILDUNGSVERZEICHNIS

---

Abbildung 2.1	Ablauf einer Web-Anfrage . . . . .	4
---------------	------------------------------------	---

## LISTINGS

---

3.1	Generische Abfrage der Dokumentenliste . . . . .	9
3.2	Sub-Abfrage pro Dokument . . . . .	10
3.3	Persistence-Kontext Statistik . . . . .	12
5.1	ein sql beispiel . . . . .	15
A.1	Zeitmessung . . . . .	19

## ABKÜRZUNGSVERZEICHNIS

---

EJB Enterprise Java Beans

JSF Java Server Faces

SFSB Stateful Session-Bean

JPA Java Persistence API

## EINLEITUNG

---

Die Akzeptanz und damit die Verwendung einer Software hängt von verschiedenen Kriterien ab. Hierbei ist neben der Stabilität und der Fehlerfreiheit die Performance beziehungsweise die Reaktionszeit der Software ein sehr wichtiges Kriterium. Hierfür muss sichergestellt werden, dass die Anwendung immer in kurzer Zeit reagiert oder entsprechende Anzeigen dargestellt um eine längere Bearbeitung anzuzeigen.

### 1.1 AUSGANGSLAGE

Die Grundlage zu dieser Arbeit bildet das DFG-Projekt "Edition der Korrespondenz Frank Wedekinds als Online-Volltextdatenbank". Die folgende Übersicht hierzu ist eine Anlehnung an [Mar23].

Die Editions- und Forschungsstelle Frank Wedekind (EFFW) wurde 1987 in der Hochschule Darmstadt gegründet. Ihr Intention ist es, den lange vernachlässigten Autor der europäischen Moderne in die öffentliche Aufmerksamkeit zu bringen. Die Publikation der »Kritischen Studienausgabe der Werke Frank Wedekinds. Darmstädter Ausgabe« wurde direkt nach der Erschließung der Wedekind-Nachlässe in Aarau, Lenzburg und München begonnen und im Jahre 2013 abgeschlossen.

Da der 1864 geborene Frank Wedekind heute zu einen der bahnbrechenden Autoren der literarischen Moderne zählt, aber bisher sehr wenig erforscht wurde, soll sich dies nun Ändern. Die nationalen und internationalen Korrespondenzen von und an Wedekind zeigen eine starke Vernetzung in der europäischen Avantgarde. Aktuell sind lediglich 710 der 3200 bekannten Korrespondenzstücke veröffentlicht worden.

Diese beinhalten substantiell das literarhistorische und kulturgeschichtliche Wissen über die Kultur zwischen 1880 und 1918, indem das überlieferte Material zum einen transkribiert editiert und zum anderen editionswissenschaftlich kommentiert wurde.

Um jenes zu verändern entstand das Projekt »Edition der Korrespondenz Frank Wedekind als Online-Volltextdatenbank«, welches bei der EFFW angesiedelt ist und als Kooperationsprojekt an der Johannes Gutenberg-Universität Mainz, der Hochschule Darmstadt und der Fernuni Hagen umgesetzt und durch die Deutsch Forschungsgemeinschaft (Bonn) gefördert wird.

Das entstandene Pilotprojekt ist eine webbasiert Anwendung, die aktuell unter <http://briefedition.wedekind.h-da.de> eingesehen werden kann. Hierbei wurden sämtliche bislang bekannte Korrespondenzen in dem System digitalisiert. Die Briefe selbst werden im etablierten TEI-Format gespeichert und über einen WYSIWYG-Editor von den Editoren und Editorinnen eingegeben.

Das Projekt wurde anhand von bekannten und etablierten Entwurfsmustern umgesetzt um eine modulare und unabhängige Architektur zu gewährleisten, damit dies für weitere digitale Briefeditionen genutzt werden kann.

## 1.2 ZIEL DER ARBEIT

Die aktuelle Umsetzung beinhaltet die bisher definierten Anforderungen vollständig, darunter fallen die Recherchemöglichkeiten, sowie auch die Eingabe und die Verarbeitung der Briefe. Ein größeres Problem hierbei ist die Performance der Oberfläche. Auf Grund der langen Abfragedauer des Datenbestandes leidet die Akzeptanz der Anwendung.

Das Ziel der Arbeit ist es, die Abfragedauer zu verringern, wodurch die Performance der Oberfläche signifikant verbessert wird.

Hierbei ist auch ein Vergleich mit anderen Technologien angedacht.

## 1.3 GLIEDERUNG

Zu Beginn der Arbeit werden im Kapitel 2 die Struktur und der grundsätzliche Aufbau der Anwendung erklärt. Hierbei wird aufgezeigt an welchen Stellen es immer wieder zu Unstimmigkeiten kommen kann und wie diese zu überprüfen sind.

Nachfolgend wird im Kapitel 3 die Konzepte vorgestellt, die die Stellen ermitteln, die eine schlechte Performance aufweisen und optimiert werden sollen. Hierbei ist zusätzlich ein Blick auf andere Frameworks sinnvoll, um dort aus den bekannten Anomalien zu lernen und deren Lösungsansatz zu überprüfen ob dieser angewandt werden kann.

Bei der Performance-Untersuchung in Kapitel 4 werden nun die Konzepte angewandt, um die Problemstellen zu identifizieren. Diese werden dann bewertet, unter den Gesichtspunkten ob eine Optimierung an dieser Stelle sinnvoll ist, oder ob der Arbeitsaufwand dafür zu enorm ist.

Nach der Entscheidung der Reihenfolge der zu bearbeitenden Punkte, wird im Kapitel 5 je nach Problemart ein gesondertes Vorgehen der Optimierung durchgeführt, um diese zu beheben oder mindestens in einen akzeptablen Rahmen zu verbessern. Diese Optimierungen werden dann in der Software entsprechend der Vorgaben angepasst.

Nach der Optimierung kommt nun die Evaluierung im Kapitel 6, um zu überprüfen ob die Anpassungen die gewünschte Verbesserung in der Performance gebracht haben.

Zum Abschluss im Kapitel 7 wird explizit die Anpassungen dargestellt, die zu einer merklichen Verbesserung geführt haben und wie diese entsprechend umgesetzt werden müssen. Zusätzlich wird beschrieben wie ein weiteres Vorgehen durchgeführt werden kann.

Da die Anwendung als Webseite umgesetzt ist, ist der zugehörige Client für den Benutzer ein Webbrowser. Dies bedeutet, dass jeder Wechsel einer Seite oder eine Suchanfrage als Web-Request an den Server geschickt wird. Solch ein Web-Request geht durch mehrere Schichten des Server-System bis die Antwort an den Client zurückgesendet wird, wie in 2.1 dargestellt.

Angefangen bei der Anfrage die über den Webbrowser an den Server gestellt wird und vom *Glassfish*-Server empfangen wird. In diesem wird anhand des definierten Routing entschieden, an welche *Java Server Page* die Anfrage weitergeleitet und verarbeitet wird. In dieser wird die Darstellung der Webseite geladen und die Anfragen für den darzustellenden Datenbestand abgeschickt.

Die Datenanfragen werden über die *EJB* an die *JPA* weitergeleitet. Hier wird nun geprüft, ob die Daten aus dem *OpenJPA Cache* direkt ermittelt werden können, oder ob die Abfrage an das unterlagerte Datenbankmanagementsystem *PostgreSQL* weitergeleitet werden muss. Die ermittelten Daten vom DBMS werden bei Bedarf im *OpenJPA Cache* aktualisiert.

Das *PostgreSQL* besteht aus mehreren Teilen die ineinander greifen um die Anfragen zu bearbeiten. Dabei sind die *Memory Buffers* notwendig um den Zugriff auf die Festplatte zu reduzieren, um die Bearbeitungszeit zu verringern. Um Anfragen die den Zugriff auf die Festplatte benötigen effizienter zu gestalten, bereiten die *Services* die Datenstrukturen auf.

## 2.1 GLASSFISH - ENTERPRISE KIRSTEN.RITZMANN@GMX.NETJAVA BEANS

In den Java-EE-Anwendungen wird der *Persistenzkontext* für die Anfragen vom *Application-Server* bereitgestellt. Hierfür werden *Application-Server* wie *GlassFish* genutzt, um die Verwendung eines Pools von Datenbankverbindungen zu definieren [MW12, S. 68]. Dadurch kann die Anzahl der Verbindung geringer gehalten werden als die Anzahl der Benutzer die an der Anwendung arbeiten. Zusätzlich werden die Transaktionen über *Stateful Session-Bean (SFSB)* gehandhabt, welche automatisch vor dem Aufruf erzeugt und danach wieder gelöscht werden. Dies birgt allerdings den Nachteil, dass der *Persistenzkontext* sehr groß werden kann, wenn viele Entities in den *Persistenzkontext* geladen werden. Da dies häufig zu Speicher- und damit Performanz-Problemen [MW12, S. 79] führen kann, muss hier darauf geachtet werden, nicht mehr benötigte Entities aus dem *Persistenzkontext* zu lösen.

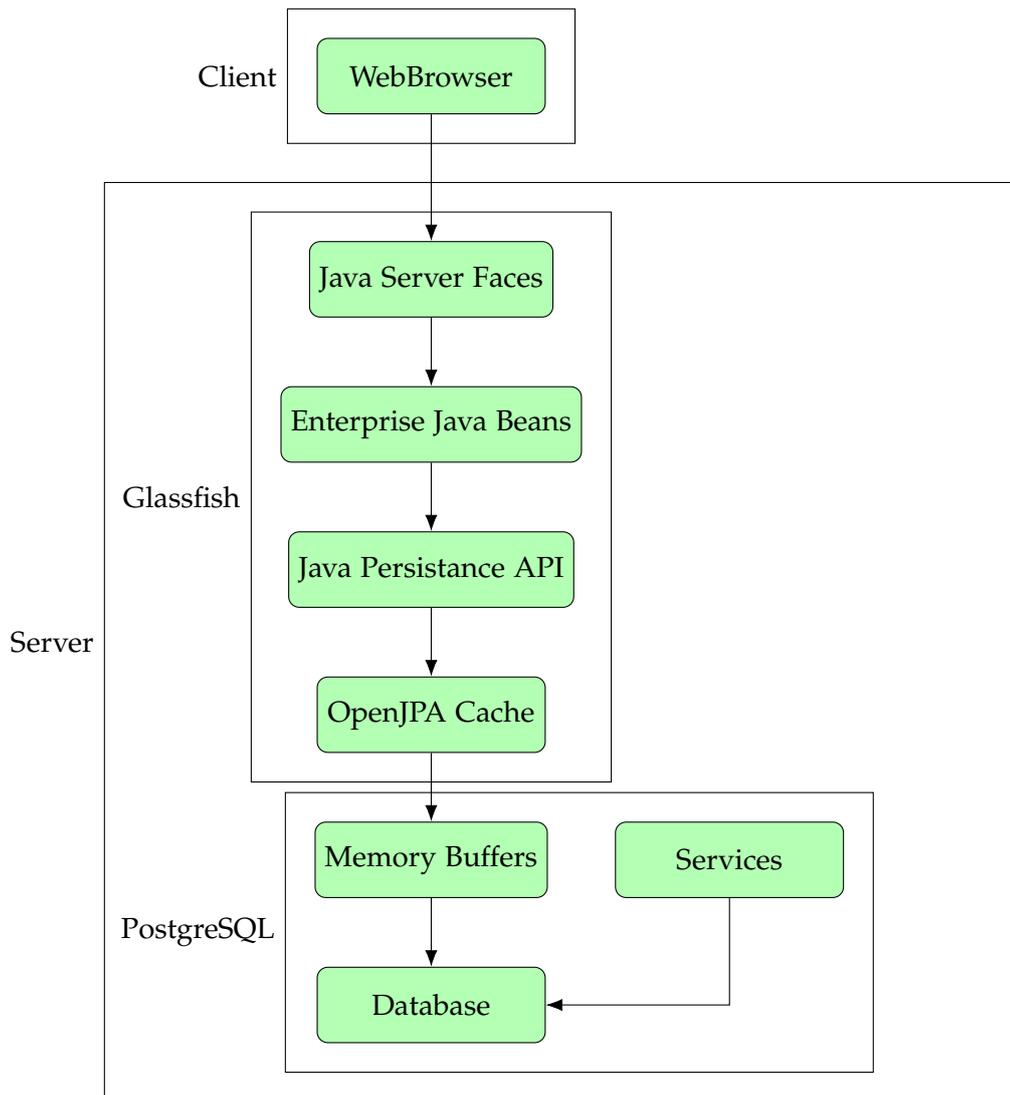


Abbildung 2.1: Ablauf einer Web-Anfrage

## 2.2 GLASSFISH - JAVA PERSISTENCE API

Die *JPA* wird als First-Level-Cache in Java-EE-Anwendung verwendet, hier nehmen die Objekte einen von vier Zuständen ein [MW<sub>12</sub>, S. 57]. Im Zustand *Transient* sind die Objekte erzeugt, aber noch nicht in den Cache überführt worden. Wenn diese in den Cache überführt worden sind, nehmen sie den Zustand *Verwaltet* ein. Ist das Objekt aus dem Cache und der Datenbank entfernt worden, nimmt es den Zustand *Gelöscht* an. *Losgelöst* ist der letzte Zustand, bei dem das Objekt aus dem Cache entfernt worden ist, aber nicht aus der Datenbank.

Eine Menge von Objekten wird als *Persistenzkontext* bezeichnet. Solange die Objekte dem *Persistenzkontext* zugeordnet sind, also den Zustand *Verwaltet* besitzen, werden diese auf Änderungen überwacht, um sie am Abschluss mit der Datenbank zu synchronisieren. In der Literatur wird hierzu der Begriff *Automatic Dirty Checking* verwendet [MW<sub>12</sub>, S. 61].

### 2.3 GLASSFISH - OPENJPA CACHE

Zusätzlich kann im *JPA* ebenfalls noch der *Second Level Cache* (L2-Cache) aktiviert werden. Dieser steht jedem *Persistenzkontext* zur Verfügung und kann dadurch die Anzahl der Datenbankzugriffe deutlich reduzieren, was bei langsamen Datenbank-Anbindungen zu hohen Performance-Gewinnen führen kann [MW12, S. 171]. Gegen die Verwendung spricht, dass die Daten im *Second Level Cache* explizit über Änderungen informiert werden müssen, welche sonst beim nächsten Aufruf veraltete Werte liefern. Ebenfalls benötigt so ein Cache einen höheren Bedarf an Arbeitsspeicher, in dem die Daten parallel zur Datenbank bereitgestellt werden, daher ist die Benutzung nur problemlos bei Entities möglich, auf die meist lesend zugegriffen wird.

In der OpenJPA-Erweiterung für den L2-Cache, wird in *Objekt-Cache* (in OpenJPA als *DataCache* bezeichnet) und *Query-Cache* unterschieden. Über die Funktionen `find()` und `refresh()` oder einer *Query* werden die geladenen Entities in den Cache gebracht. Davon ausgenommen sind *Large Result Sets* (Abfragen die nicht alle Daten auf einmal laden), *Extent-Technologien* und *Queries*, die einzelne Attribute von Entities zurückliefern, aber nicht das Entity selbst. Hierbei kann genau gesteuert werden, welche Entity in den Cache abgelegt wird und welche nicht. Ebenfalls kann auf Klassenbasis der zugehörige Cache definiert werden, um eine bessere Last-Verteilung beim Zugriff zu ermöglichen [MW12, S. 314].

Im *Query-Cache* werden die Abfragen bzw. die Eigenschaften einer Abfrage und die zurückgelieferten Ids der Entities gespeichert. Bei einem erneuten Aufruf dieser Abfrage werden die referenzierten Objekte aus dem *Objekt-Cache* zurückgegeben. Bei veränderten referenzierten Entities wird der *Query-Cache* nicht genutzt und die betroffenen Abfragen werden unverzüglich aus dem *Query-Cache* entfernt [MW12, S. 316].

Um zu prüfen, ob die Einstellungen sinnvoll gesetzt sind, kann in OpenJPA eine Cache-Statistik abgefragt werden. Mit dieser kann die Anzahl der Lese- und Schreibzugriffe im Cache überprüft werden, entsprechend dieser Auswertung sollten die Einstellungen an den Entities angepasst werden [Ibm].

### 2.4 POSTGRESQL - MEMORY BUFFERS

Die Speicherverwaltung des PostgreSQL-Servers muss für Produktivsysteme angepasst werden [EH13, S. 34–38]. Hierunter fallen die *shared\_buffers* die bei ca. 10 bis 25 Prozent des verfügbaren Arbeitsspeichers liegen sollten. Mit dieser Einstellung wird das häufige Schreiben des Buffers durch Änderungen von Daten und Indexen auf die Festplatte reduziert.

Die Einstellung *temp\_buffers* definiert wie groß der Speicher für temporäre Tabellen pro Verbindung maximal werden darf und sollte ebenfalls überprüft werden. Ein zu kleiner Wert bei großen temporären Tabellen führt zu einem signifikanten Leistungseinbruch, wenn die Tabellen nicht im Hauptspeicher, sondern in einer Datei ausgelagert werden.

Der *work\_mem* definiert die Obergrenze des zur Verfügung gestellt Hauptspeichers pro Datenbankoperation wie effizientes Sortieren, Verknüpfen oder Filtern. Ebenso wird im Falle eines zu klein gewählten Speichers auf temporäre Dateien auf der Festplatte ausgewichen, was signifikanten Leistungseinbrüchen zur Folge haben kann.

Die *maintenance\_work\_mem* wird bei Verwaltungsoperationen wie Änderungen und Erzeugungen von Datenbankobjekten als Obergrenze definiert. Die Wartungsaufgabe *VACUUM*, welche die fragmentierten Tabellen aufräumt und somit die Performance hebt, beachtet die Obergrenze ebenfalls.

## 2.5 POSTGRESQL - SERVICES

Die Wartung des Datenbanksystems ist eine der wichtigsten Aufgaben und sollte regelmäßig durchgeführt werden, damit die Performance des Systems durch die Änderungen des Datenbestands nicht einbricht [EH13, S. 75]. Hierfür gibt es den *VACUUM*-Befehl, welcher entweder per Hand oder automatisch durch das Datenbanksystem ausgeführt werden soll. Für die automatische Ausführung kann der maximal verwendete Speicher über die Einstellung *autovacuum\_work\_mem* gesondert definiert werden [Posa]. Neben dem Aufräumen durch *VACUUM*, sollten auch die Planerstatistiken mit *ANALYZE* [EH13, S. 83] aktuell gehalten werden, damit die Anfragen durch den Planer richtig optimiert werden können. Für beide Wartungsaufgaben gibt es den *Autovacuum*-Dienst, dieser sollte aktiv und richtig konfiguriert sein.

Mit dem Tool *pgFouine* [EH13, S. 155] können die Logs des PostgreSQL Server analysiert und auf Probleme hin untersucht werden. Hiermit können sehr einfach die häufigsten bzw. langsamsten Anfragen ermittelt werden.

## 2.6 POSTGRESQL - ABFRAGEN

Für weitere Optimierungen werden anschließend die Anfragen einzeln überprüft. Hierfür ist es sinnvoll die Ausführungspläne der Abfrage zu analysieren [EH13, S. 252], die verschiedenen Plantypen und ihre Kosten zu kennen, sowie die angegebenen Werte für die Plankosten zu verstehen [DNB21, S. 24–30]. Besonderes Augenmerk gilt dem Vergleichen des tatsächlich ausgeführten mit dem ursprünglichen Plan [EH13, S. 254]. Eine der wichtigsten Kennzeichen hierbei ist, ob die Zeilenschätzung akkurat war, größere Abweichungen weisen häufig auf veraltete Statistiken hin.

Um die Abfragen selbst zu optimieren, gibt es ein Vorgehen über mehrere Schritte [DNB21, S. 304–308]. Zuerst wird Unterschieden, ob es sich um eine *Kurze* oder eine *Lange* Abfrage handelt. Im Falle einer *Kurzen* Abfrage, werden zuerst die Abfragekriterien überprüft. Sollte dies zu keiner Verbesserung führen, werden die Indexe geprüft. Ist dies ebenso erfolglos, wird die Abfrage nochmals genauer analysiert und so umgestellt, dass die restriktivste Einschränkung zuerst zutrifft. Bei einer *Langen* Abfrage soll überprüft werden, ob es sinnvoll ist, das Ergebnis in einer Tabelle zu speichern und bei Änderungen zu aktualisieren. Wenn dies nicht möglich ist, sollten die

folgenden Schritte durchgeführt werden. Zuerst wird der restriktivste Join gesucht und überprüft, ob dieser als Erstes ausgeführt wird. Anschließend fügt man weitere Joins hinzu und prüft die Ausführungszeit und die Abfragepläne. Als Nächstes wird sich vergewissert, ob große Tabellen nicht mehrfach durchsucht worden sind. Bei Gruppierungen ist noch zu prüfen, ob diese früher durchgeführt werden können, um die Abfragemenge zu verringern.

Bei *Langen* Abfragen ist die Abhandlung »Optimizing Iceberg Queries with Complex Joins« [WRY17] ein zusätzlicher Ratgeber, um die Performance zu steigern.

Des Weiteren können über das Modul `pg_stat_statements` Statistiken der Aufrufe die an den Server gestellt wurden, ermittelt werden [Posb]. Hierbei können die am häufigsten Aufgerufenen und die Anfragen mit der längsten Ausführungszeit ermittelt werden.

## KONZEPT

---

Das folgende Kapitel enthält die im Rahmen dieser Arbeit entstandenen Konzepte, um die vorhandenen Probleme zu identifizieren und mit entsprechenden Maßnahmen entgegenzusteuern. Hierbei werden zum einen die Konfigurationen der eingesetzten Software überprüft. Zum anderen werden die verschiedenen Schichten der entwickelten Software auf mögliche Optimierungen untersucht und bewertet.

### 3.1 ALLGEMEINE BETRACHTUNG DES SYSTEMS

Für die Untersuchung des Systems wird der direkte Zugang zum Server benötigt. Hierbei werden zuerst die im Kapitel 2.5 beschriebenen Einstellungen überprüft.

Zuerst wird am PostgreSQL-Server die Konfiguration der Speicher mit der Vorgabe für Produktivsystem abgeglichen. Hierunter fallen die Einstellungen für die *shared\_buffers*, der bei einem Arbeitsspeicher von mehr als 1 GB ca. 25% des Arbeitsspeicher definiert sein soll [Posc].

*TODO: die anderen Speicher abarbeiten?*

Dann wird mit dem Systemtools, wie den Konsolenanwendungen *htop* und *free*, die Auslastung des Servers überprüft. Hierbei ist die CPU-Leistung, der aktuell genutzte Arbeitsspeicher, sowie die Zugriffe auf die Festplatte die wichtigen Faktoren zur Bewertung.

Die CPU-Leistung sollte im Schnitt nicht die 70% überschreiten, für kurze Spitzen wäre dies zulässig. Da sonst der Server an seiner Leistungsgrenze arbeitet und dadurch es nicht mehr schafft die gestellten Anfragen schnell genug abzuarbeiten.

Da unter Linux der Arbeitsspeicher nicht mehr direkt freigegeben wird, ist hier die Page-Datei der wichtigere Indikator. Wenn dieses in Verwendung ist, dann benötigen die aktuell laufenden Programme mehr Arbeitsspeicher als vorhanden ist, wodurch der aktuell nicht verwendete in die Page-Datei ausgelagert wird. Hierdurch erhöhen sich die Zugriffszeiten auf diese Elemente drastisch.

Die Zugriffsgeschwindigkeit, die Zugriffszeit sowie die Warteschlange an der Festplatte zeigt deren Belastungsgrenze auf. Hierbei kann es mehrere Faktoren geben. Zum einem führt das Paging des Arbeitsspeicher zu erhöhten Zugriffen. Ein zu klein gewählter Cache oder gar zu wenig Arbeitsspeicher erhöhen die Zugriffe auf die Festplatte, da weniger zwischengespeichert werden kann und daher diese Daten immer wieder direkt von der Festplatte geladen werden müssen.

## 3.2 UNTERSUCHUNG DER ANWENDUNG

Bei der Performance-Untersuchung der Anwendung, wird sich im ersten Schritt auf die Dokumentenliste beschränkt. Anhand dieser können die Optimierungen getestet und überprüft werden. Im Nachgang können die daraus gewonnenen Kenntnisse auf die anderen Abfragen übertragen werden.

Die Dokumentenliste zeigt direkte und indirekte Informationen zu einem Dokument an. Hierzu gehört die Kennung des Dokumentes, das Schreibdatum, der Autor, der Adressat, der Schreibort und die Korrespondenzform. Nach jeder dieser Information kann der Bediener die Liste auf- oder absteigend sortieren lassen. Zusätzlich wird die Liste immer nach dem Schreibdatum sortiert, um die Ergebnisse bei gleichen Werten der zu sortierenden Informationen, wie dem Schreibort, immer in einer chronologisch aufsteigenden Form zu darzustellen.

Aktuell verwenden die Editoren die Dokumentenliste um die Briefe eines Adressaten zu filtern und diese in chronologische Reihenfolge aufzulisten und zu untersuchen wie Kommunikation zwischen Herrn Wedekind und dem Adressaten abgelaufen ist. Ebenso wird nach Standorten sortiert, um zu prüfen mit welchen Personen sich an den ...

*TODO: Hier noch mehr Infos dazu, für was genau die Editoren diese tun*

Da die Daten in der 3. Normalform in der Datenbank gespeichert werden, sind einige Relationen für die Abfragen notwendig. Dies wird durch die generische Abfrage in Listing 3.1 gezeigt. Zusätzlich wird für jedes dargestellte Dokument eine zusätzliche Abfrage durchgeführt, die in Listing 3.2 zeigt, dass auch hier weitere Relationen notwendig sind.

Listing 3.1: Generische Abfrage der Dokumentenliste

---

```
SELECT DISTINCT
-- document
  t0.id, t0.createdat, t0.modifiedat, t0.validuntil, t0.envelope_id
, t0.firstprint_id, t0.followsdocument_id, t0.iscomplete
, t0.isdispatched, t0.ispublishedindb, t0.isreconstructed
, t0.location_id, t0.numberofpages, t0.numberofsheets
, t0.parentdocument_id, t0.reviewer_id, t0.signature, t0.bequest_id
, t0.city_id, t0.documentcategory, t0.documentcontentteibody
, t0.datetype, t0.enddatestatus, t0.endday, t0.endmonth, t0.endyear
, t0.startdatestatus, t0.startday, t0.startmonth, t0.startyear
, t0.documentdelivery_id, t0.documentid, t0.documentstatus
-- historical person
, t4.id, t4.createdat, t4.modifiedat, t4.validuntil, t4.firstname
, t4.surname, t4.title, t4.birthdatetype, t4.birthstartdatestatus
, t4.birthstartday, t4.birthstartmonth, t4.birthstartyear
, t4.birthendstatus, t4.birthendday, t4.birthendmonth, t4.birthendyear
, t4.deathdatetype, t4.deathstartdatestatus, t4.deathstartday
, t4.deathstartmonth, t4.deathstartyear, t4.deathendstatus
, t4.deathendday, t4.deathendmonth, t4.deathendyear, t4.dnbref
, t4.gender, t4.isinstitution, t4.personid, t4.pseudonym, t4.wikilink
-- extended biography
, t5.id, t5.createdat, t5.modifiedat, t5.validuntil
```

```

-- sitecity birth
, t6.id, t6.createdat, t6.modifiedat, t6.validuntil
, t6.extendedbiography_id, t6.city, t6.country, t6.dnbref, t6.region
, t6.sitecityid, t6.wikilink, t6.zipcode
-- sitecity death
, t7.id, t7.createdat, t7.modifiedat, t7.validuntil
, t7.extendedbiography_id, t7.city, t7.country, t7.dnbref, t7.region
, t7.sitecityid, t7.wikilink, t7.zipcode
-- sitecity
, t8.id, t8.createdat, t8.modifiedat, t8.validuntil
, t8.extendedbiography_id, t8.city, t8.country, t8.dnbref, t8.region
, t8.sitecityid, t8.wikilink, t8.zipcode
-- appuser
, t9.id, t9.createdat, t9.modifiedat, t9.validuntil, t9.activated
, t9.emailaddress, t9.firstname, t9.institution, t9.lastlogindate
, t9.loggedin, t9.loggedinsince, t9.loginname, t9.password
, t9.registrationdate, t9.salt, t9.surname, t9.title
-- appuserrole
, t10.id, t10.createdat, t10.modifiedat, t10.validuntil
, t10.description, t10.userrole
FROM public.Document t0
LEFT OUTER JOIN public.DocumentCoAuthorPerson t1 ON t0.id = t1.
    document_id
LEFT OUTER JOIN public.DocumentAddresseePerson t2 ON t0.id = t2.
    document_id
LEFT OUTER JOIN public.historicalperson t3 ON t0.authorperson_id = t3.
    id
LEFT OUTER JOIN public.historicalperson t4 ON t0.authorperson_id = t4.
    id
LEFT OUTER JOIN public.sitecity t8 ON t0.city_id = t8.id
LEFT OUTER JOIN public.appuser t9 ON t0.editor_id = t9.id
LEFT OUTER JOIN public.extendedbiography t5 ON t4.extendedbiography_id
    = t5.id
LEFT OUTER JOIN public.sitecity t6 ON t4.sitecity_birthe_id = t6.id
LEFT OUTER JOIN public.sitecity t7 ON t4.sitecity_death_id = t7.id
LEFT OUTER JOIN public.appuserrole t10 ON t9.appuserrole_id = t10.id
WHERE (t0.validuntil > NOW()
    AND t0.ispublishedindb = true
    AND (t1.validuntil > NOW() OR t1.id IS NULL)
    AND (t2.validuntil > NOW() OR t2.id IS NULL)
    AND 1 = 1
    )
ORDER BY t0.startyear DESC, t0.startmonth DESC, t0.startday DESC
LIMIT 400

```

Listing 3.2: Sub-Abfrage pro Dokument

```

SELECT
-- document coauthor person
    t0.id, t0.createdat, t0.modifiedat, t0.validuntil, t0.document_id
, t0.status
-- historical person

```

```

, t1.personid, t1.pseudonym, t1.wikilink, t1.id, t1.createdat
, t1.modifiedat, t1.validuntil, t1.comments, t1.firstname, t1.surname
, t1.title, t1.birthdatetype, t1.birthstartdatestatus
, t1.birthstartday, t1.birthstartmonth, t1.birthstartyear
, t1.birthendstatus, t1.birthendday, t1.birthendmonth, t1.birthendyear
, t1.deathdatetype, t1.deathstartdatestatus, t1.deathendstatus
, t1.deathstartday, t1.deathstartmonth, t1.deathstartyear
, t1.deathendday, t1.deathendmonth, t1.deathendyear
, t1.dnbref, t1.gender, t1.isinstitution
-- extended biography
, t2.id, t2.createdat, t2.modifiedat, t2.validuntil, t2.description
-- sitecity birth
, t3.id, t3.createdat, t3.modifiedat, t3.validuntil
, t3.extendedbiography_id, t3.city, t3.country, t3.dnbref, t3.region
, t3.sitecityid, t3.wikilink, t3.zipcode
-- sitecity death
, t4.id, t4.createdat, t4.modifiedat, t4.validuntil
, t4.extendedbiography_id, t4.city, t4.country, t4.dnbref, t4.region
, t4.sitecityid, t4.wikilink, t4.zipcode
FROM public.DocumentCoAuthorPerson t0
LEFT OUTER JOIN public.historicalperson t1 ON t0.authorperson_id = t1.
id
LEFT OUTER JOIN public.extendedbiography t2 ON t1.extendedbiography_id
= t2.id
LEFT OUTER JOIN public.sitecity t3 ON t1.sitecity_birth_id = t3.id
LEFT OUTER JOIN public.sitecity t4 ON t1.sitecity_death_id = t4.id
WHERE t0.document_id = ?

```

Nach aktuellem Stand beinhaltet die Datenbank ca. 5400 Briefe, für die jeweils 2-7 eingescannte Faksimile gespeichert werden. Diese Graphik-Dateien werden im TIFF-Format abgespeichert und benötigen zwischen 1 und 80 MB Speicherplatz. Dadurch kommt die Datenbank aktuell auf ca. 3,8 GB.

*TODO: Die unteren Punkte nochmal untergliedern in Performance-messen und Statistiken prüfen, dann kann mit den Performance-messen die unterschiedlichen Aktionen durchgeführt werden in Kapitel 4*

Wie im Kapitel 2 dargestellt, besteht die eigentliche Anwendung aus mehreren Schichten. Die PostgreSQL-Schicht wurde schon im vorherigen Kapitel betrachtet. Daher gehen wir nun weiter nach oben in den Schichten vom Glassfish-Server.

Die OpenJPA Cache Schicht wird nun einzeln untersucht. Hierfür werden die zuerst die Cache-Statistik für Object-Cache und Query-Cache aktiviert [MW12, S. 315]. Die somit erfassten Werte, werden über eine Webseite bereitgestellt, um die Daten Live vom Server verfolgen zu können. Zusätzlich können diese Daten über ein Skript zyklisch abgefragt, gespeichert und verglichen werden.

In der JPA Schicht sind die Anzahl der Entitäten im Persistence Context zu beobachten. Die Anzahl der verschiedenen Klassen soll ermittelt und die Statistik-Webseite um diese Daten erweitern. Um die Daten zu ermitteln, kann der Quellcode aus 3.3 verwendet werden.

Listing 3.3: Persistence-Kontext Statistik

---

```

EntityManagerFactory emf = Persistence.createEntityManagerFactory(...);
EntityManager em = emf.createEntityManager();
for(EntityType<?> entityType : em.getMetaModel().getEntities())
{
    Class<?> managedClass = entityType.getBindableJavaType();
    System.out.println("Managing type: " + managedClass.getCanonicalName
        ());
}

// Oder bei JPA 2.0
emf.getCache().print();

```

---

Die Schicht EJB besitzt keine Möglichkeit um eine sinnvolle Messung durchzuführen, daher wird hierfür keine direkte Messungen eingefügt.

Bei den Java Server Faces (JSF) wird eine Zeitmessung eingefügt. Hierfür müssen die Seiten so erweitert werden, dass zum einen die Zeit gemessen wird um die Daten zu ermitteln. Zum anderen wird die Zeit gemessen wie lange es dann noch dauert um die Seite mit den Daten zu rendern um diese an den Client auszuliefern. Diese 2 Zeiten sollen dann im Footer direkt auf der Seite mit dargestellt werden. Somit kann der Benutzer auch direkt sehen, wenn das laden länger gedauert hat, an welcher Stelle die Verzögerung aufgetreten ist.

Die Abfragen werden ebenfalls untersucht und mit verschiedenen Methoden optimiert. Hierfür werden zum einen auf native SQL-Anfragen umgestellt und die Ausführungszeiten überprüft. Ebenfalls werden die Abfragen durch Criteria API erzeugt und dessen Ausführungszeit ermittelt.

Zusätzlich werden im SQL-Server Optimierungen vorgenommen, darunter zählen die materialized views, welche eine erweiterte View sind. Neben der Abfrage der Daten beinhaltet diese auch noch nicht vorberechneten Daten der Abfrage, womit diese viel schneller abgefragt werden können. Zusätzlich werden die cached queries überprüft ob diese eine Verbesserung der Performance und der Abfragedauern verkürzen können.

Um die Optimierungen in der Anwendung zu überprüfen, werden die Webseiten über ein Shell-Skript abgefragt. Das Skript ruft automatisiert die URLs der Webseite mehrfach ab. Die Dauer der Aufrufe der Webseiten werden gemessen und statistisch ausgewertet. Für einen späteren Vergleich werden diese Informationen gesichert und mit einem erneuten Aufruf nach den Optimierungen verglichen. Hierdurch kann auch festgestellt werden, ob die Optimierungen erfolgreich waren. Um die Netzwerklatenz ignorieren zu können, wird das Skript auf dem gleichen Computer aufgerufen, auf dem die Webseite gestartet wurde.

Das zugehörige Script ist im Anhang A angehängt.

## PERFORMANCE-UNTERSUCHUNG

---

### 4.1 AUSWERTUNG DES SYSTEMS

*TODO: Hier die Auswertung des Produktionsservers unterbringen*

### 4.2 STATISTIKEN IM POSTGRESQL AUSWERTEN

*TODO: Logs auswerten, am besten vom Produktionsserver. Ebenfalls sollte man die Webseite prüfen, die den Cache von OpenJPE auswerten*

### 4.3 ÜBERPRÜFUNG DES POSTGRESQL UND SERVERS

*TODO: Konfiguration vom Produktionsserver prüfen*

### 4.4 EINBAU UND AKTIVIEREN VON PERFORMANCE-MESSUNG

*TODO: Einbau der Messungen direkt in die Webseite bzw. in ein Log*

### 4.5 UNTERSUCHUNG DER ANWENDUNG

Nun werden die unterschiedlichen Schichten betrachtet und möglichen Performance-Verbesserungen untersucht und deren Vor- und Nachteile herausgearbeitet.

*TODO: Bei deaktivierten Daten und Query-Cache hat der 3te Aufruf im Durchschnitt schon doppelt so lange gedauert wie die 2 vorherigen und beim 4ten Aufruf ist der Payara-Server mit einer OutOfMemoryError-Exception im Java-Heapspeicher ausgestiegen und musste per Kill-Befehl abgeschossen werden. Angeblich liegt es am der Speicher-Option, in der domain1/config/domain.xml die Einstellung `<jvm-options>-Xmx512m</jvm-options>` gesucht und den Inhalt durch `-Xmx4096m` ersetzt. Nach dem ändern der Konfiguration ist der Absturz auch nach dem 30 Aufruf des Skriptes nicht reproduzierbar gewesen. Nach dem zurückstellen auf 512 konnte man den Absturz nach relativ kurze Zeit erneute erzwingen (6ter Aufruf). Der Arbeitsspeicher-Verbraucht der Anwendung lag aber bei ca. 1500MB PSS*

Unterschied RSS und PSS erklären? PSS: non-swapped physical memory + anteiliger shared memory, RSS: non-swapped physical memory + shared memory, siehe man ps: <https://man7.org/linux/man-pages/man1/ps.1.html>

Bei 10 Aufrufen waren es ca. 50 MB mehr und bei 20 Aufrufen waren es ca. 100 MB mehr Arbeitsspeichernutzung. Nach umstellung auf 4096 ist der Ser-

ver nach bei ca. 4700 MB RSS ausgestiegen. Aber ohne OutOfMemoryError und der Arbeitsspeicher im ganzen war zur Hälfte benutzt.

#### 4.5.1 *Caching im OpenJPA*

Die Cache-Einstellung von OpenJPA werden über die zwei Einstellungen `openjpa.DataCache` und `openjpa.QueryCache` konfiguriert. Bei beiden Einstellungen kann zuerst einmal über ein einfaches Flag *true* und *false* entschieden werden ob der Cache aktiv ist. Zusätzlich kann über das Schlüsselwort *CacheSize* die Anzahl der Elementen im Cache gesteuert werden. Wird diese Anzahl erreicht, dann werden zufällige Objekte aus dem Cache entfernt und in eine `SoftReferenceMap` übertragen. Dort

Bei aktivierten Caching benötigt der erste Aufruf ca. 1500 msec, die nachfolgenden liegen um die 250 msec.

*TODO: pin und unpin noch mit einbringen? SoftReferenceMap nochmal genau durchleuchte, laut doku entfällt dort nichts wenn kein Timeout auf der Klasse definiert ist*

Kein unterschied feststellbar, nachgeprüft ob die Einstellung richtig deaktiviert sind in der Datei: `domain1/applications/WedekindJSF-1.0.0/WEB-INF/lib/classes/META-INF/persistence.xml`

#### 4.5.2 *Caching im JPA*

#### 4.5.3 *Caching in EJB*

#### 4.5.4 *Abfragen JPQL*

#### 4.5.5 *Abfragen Criteria API*

#### 4.5.6 *materialized views*

#### 4.5.7 *cached queries*

#### 4.5.8 *Umgestalten der Datenbanktabellen*

#### 4.5.9 *Verkleinerung der Abfragen*

??OPTIMIERUNG??

---

*TODO: Muss noch entsprechend der Auswertungen aus der Performance-Untersuchungen angepasst werden*

5.1 ERMITTLUNG DER PERFORMANCE-PROBLEME

5.2 ANALYSE DER ABFRAGE

5.3 OPTIMIERUNGEN DER ABFRAGEN

5.4 ANPASSUNG DER KONFIGURATION

und hier ein sql-beispiel Listing 5.1

Listing 5.1: ein sql beispiel

---

```
select *  
from tblCPDataX  
where szName = N'EDA01'
```

---

## EVALUIERUNG

---

*TODO: Hier noch darauf verweisen, dass eine Befragung unter den Benutzer und Entwickler nicht zielführend gewesen wäre, da zu wenige Anwender, 4 Stück, daher ist der rein technische Ansatz die einzige sinnvolle Wahl*

*TODO: Zusätzlich beschreiben welche Möglichkeiten man genau genutzt hat und warum bzw. warum nicht*

6.1 ERNEUTE LAUFZEITANALYSE STARTEN

6.2 STATISTIKEN IM POSTGRESQL AUSWERTEN

6.3 VERGLEICH DER ERGEBNISSE VOR UND NACH DER OPTIMIERUNG

ZUSAMMENFASSUNG UND AUSBLICK

---

Teil I

APPENDIX



## ZEITMESSUNG DER WEBSEITE

---

Mit dem nachfolgenden Skript werden die hinterlegten URLs mehrfach ausgeführt. Jeder Aufruf wird gemessen und pro URL die kürzeste, die längste, die durchschnittliche Laufzeit und die Standardabweichung ausgegeben.

Listing A.1: Zeitmessung

---

```
#!/bin/bash
#

# Activate Bash Strict Mode
set -euo pipefail

main() {
    {
        echo -e "URL\tRuns\tStDev\tMin (ms)\tAvg (ms)\tMax (ms)"
        for url in ${@:2}; do
            get_statistics $url $1
        done
    } #| column -s $'\t' -t
}

get_statistics() {
    # Initialize the variables
    local min=1000000000
    local max=0
    local dur=0
    local durQ=0
    local spin=0
    local spinner=('-' '\ ' '| ' '/' )

    echo -ne "$1\t "

    # repeat for the defined counts the url calling
    for i in $(seq 1 $2); do
        echo -ne "\b${spinner[$spin]}"
        spin=$(( (spin+1) % 4 ))
        local gp=$((get_posts $1)/1000000) # from ns to ms
        if [[ $gp -gt $max ]]
            then max=$gp
        fi
        if [[ $gp -lt $min ]]
            then min=$gp
        fi
        dur=$(( $dur + $gp ))
        durQ=$(( $durQ + (($gp * $gp) )) )
    done
}
```

```

done

local avg=$((dur/$2))
local avgPow=$((avg * avg))
local stdev=$( echo "sqrt(($durQ / $2) - $avgPow)" | bc )

# output the statistic values
echo -e "\b$2\t$stdev\t$min\t$avg\t$max"
}

get_posts() {
# Call the url with measure time
local start=$(date +%s%N)
curl --silent --show-error $1 > /dev/null
local stop=$(date +%s%N)
local dur=$((stop-$start))
echo $dur
}

print_process() {
ps -C java --no-headers --format "pid %cpu %mem rss pss cmd" |\
awk 'BEGIN { printf "%6s %5s %4s %13s %13s %6-50s\n", "pid", "%cpu", "%mem", "rss Mb", "pss Mb", "cmd"}
{hr=$4/1024; hp=$5/1024; printf("%6i %5.1f %4.1f %13.2f %13.2f %s\n", $1, $2, $3,hr, hp, $6) }'
}

# the main domain
#hostname="https://briefedition.wedekind.h-da.de"
hostname="http://localhost:8080/WedekindJSF-1.0.0"

# the Array of the Urls
url_arr=(
"$hostname/index.xhtml"
"$hostname/view/document/list.xhtml"
#"$hostname/view/correspondent/list.xhtml"
#"$hostname/view/person/list.xhtml"
)

print_process
echo ""

# Execute all the URLs for 10 rounds
main 10 ${url_arr[@]}

echo ""
print_process

```

---

## LITERATUR

---

- [Ibm] 2023. URL: <https://www.ibm.com/docs/de/was/8.5.5?topic=applications-configuring-openjpa-caching-improve-performance> (besucht am 24.09.2023).
- [Posa] 2023. URL: <https://postgrespro.com/docs/postgresql/14/runtime-config-resource> (besucht am 27.12.2023).
- [Posb] 2023. URL: <https://www.postgresql.org/docs/8.4/pgstatstatements.html> (besucht am 27.12.2023).
- [Posc] 2024. URL: <https://www.postgresql.org/docs/current/runtime-config-resource.html> (besucht am 27.03.2024).
- [DNB21] Henrietta Dombrovskaya, Boris Novikov und Anna Bailliekova. *PostgreSQL Query Optimization - The Ultimate Guide to Building Efficient Queries*. Berkeley, CA: Apress, 2021. ISBN: 978-1-4842-6885-8. DOI: 10.1007/978-1-4842-6885-8. eprint: <https://link.springer.com/book/10.1007/978-1-4842-6885-8>. URL: <https://link.springer.com/book/10.1007/978-1-4842-6885-8>.
- [EH13] Peter Eisentraut und Bernd Helmle. *PostgreSQL-Administration* -. Köln: O'Reilly Germany, 2013. ISBN: 978-3-868-99362-2.
- [Mar23] AK Prof. Dr. Ariane Martin. *Frank Wedekind | FB 05 - AK Prof. Dr. Ariane Martin*. 2023. URL: <https://www.martin.germanistik.uni-mainz.de/forschung/frank-wedekind/> (besucht am 24.09.2023).
- [MW12] Bernd Müller und Harald Wehr. *Java Persistence API 2*. München: Carl Hanser Verlag GmbH & Co. KG, 2012. DOI: 10.3139/9783446431294. eprint: <https://www.hanser-elibrary.com/doi/pdf/10.3139/9783446431294>. URL: <https://www.hanser-elibrary.com/doi/abs/10.3139/9783446431294>.
- [WRY17] Brett Walenz, Sudeepa Roy und Jun Yang. "Optimizing Iceberg Queries with Complex Joins". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: Association for Computing Machinery, 2017, 1243–1258. ISBN: 9781450341974. DOI: 10.1145/3035918.3064053. URL: <https://doi.org/10.1145/3035918.3064053>.