



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

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, 21. August 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 | 4 |
| 2.1 | Glassfish - Enterprise Java Beans | 4 |
| 2.2 | Glassfish - Java Persistence API | 5 |
| 2.3 | Glassfish - OpenJPA Cache | 6 |
| 2.4 | PostgreSQL - Memory Buffers | 6 |
| 2.5 | PostgreSQL - Services | 7 |
| 2.6 | PostgreSQL - Abfragen | 7 |
| 3 | Konzept | 9 |
| 3.1 | Allgemeine Betrachtung des Systems | 9 |
| 3.2 | Untersuchung der Anwendung | 10 |
| 4 | Performance-Untersuchung | 15 |
| 4.1 | Auswertung des Systems | 15 |
| 4.2 | Statistiken im PostgreSQL auswerten | 15 |
| 4.3 | Überprüfung des PostgreSQL und Servers | 15 |
| 4.4 | Einbau und Aktivieren von Performance-Messung | 15 |
| 5 | Performance-Untersuchung der Anwendung | 18 |
| 5.1 | Umgestalten der Datenbanktabellen | 20 |
| 5.2 | Caching im OpenJPA | 20 |
| 5.3 | cached queries | 22 |
| 5.4 | Caching im JPA | 22 |
| 5.5 | Caching in EJB | 22 |
| 5.6 | Abfragen JPQL | 23 |
| 5.7 | Abfragen Criteria API | 24 |
| 5.8 | materialized views | 25 |
| 5.9 | Statische Webseiten | 31 |
| 5.10 | Client basierte Webseiten | 31 |
| 6 | Evaluierung | 32 |
| 6.1 | Nutzerumfrage | 32 |
| 6.2 | Statische Webseiten | 32 |
| 6.3 | Client basierte Webseiten | 32 |
| 6.4 | Statistiken im PostgreSQL auswerten | 33 |
| 6.5 | Vergleich der Ergebnisse vor und nach der Optimierung | 33 |
| 6.5.1 | Caching im OpenJPA | 33 |
| 6.5.2 | Cached Queries | 34 |
| 6.5.3 | Caching in JPA | 34 |
| 6.5.4 | Caching in EJB | 34 |
| 6.5.5 | Abfragen mit JPQL und Criteria API | 34 |
| 6.5.6 | Materialized View | 35 |

| | | |
|----------|------------------------------------|----|
| 7 | Zusammenfassung und Ausblick | 36 |
| I | Appendix | |
| A | Zeitmessung der Webseite | 38 |
| B | Docker Konfiguration | 41 |
| C | Aufruf Skript | 43 |
| D | JSF Performance Statistics Servlet | 49 |
| E | JSF Performance Measure | 51 |
| | Literatur | 53 |

ABBILDUNGSVERZEICHNIS

| | | |
|---------------|------------------------------------|---|
| Abbildung 2.1 | Ablauf einer Web-Anfrage | 5 |
|---------------|------------------------------------|---|

TABELLENVERZEICHNIS

| | | |
|--------------|-------------------------------------------------------------------------------|----|
| Tabelle 5.1 | Messung ohne Caches | 19 |
| Tabelle 5.2 | Messung ohne Caches im Docker | 20 |
| Tabelle 5.3 | Messung mit OpenJPA-Cache und Größe auf 1000 . . . | 21 |
| Tabelle 5.4 | Messung mit OpenJPA-Cache und Größe auf 10000 . . | 21 |
| Tabelle 5.5 | Messung mit OpenJPA-Cache und Größe auf 1000 und o SoftReference | 21 |
| Tabelle 5.6 | Messung mit aktiviertem Cached Queries | 22 |
| Tabelle 5.7 | Messung mit Enterprise Java Beans (EJB)-Cache | 22 |
| Tabelle 5.8 | Messung mit Criteria-API ohne Cache | 25 |
| Tabelle 5.9 | Messung mit Materialized View | 28 |
| Tabelle 5.10 | Messung mit erweiterter Materialized View | 29 |

LISTINGS

| | | |
|-----|--------------------------------------------------|----|
| 3.1 | Generische Abfrage der Dokumentenliste | 10 |
| 3.2 | Sub-Abfrage pro Dokument | 12 |
| 3.3 | Persistence-Kontext Statistik | 13 |
| 4.1 | Ermitteln der PostgreSQL Einstellungen | 15 |
| 4.2 | Einbindung Factory | 16 |
| 4.3 | PostgreSQL Dateikonfiguration | 16 |
| 4.4 | PostgreSQL Ausgabekonfiguration | 16 |
| 5.1 | JPQL Dokumentenliste | 23 |
| 5.2 | Java JPQL Dokumentenliste | 23 |
| 5.3 | Criteria API Dokumentenliste | 24 |
| 5.4 | SQL Materialized View | 26 |
| 5.5 | SQL Materialized View | 27 |
| 5.6 | SQL Materialized View Erweiterung | 28 |
| 5.7 | DataTable mit Json | 29 |
| 5.8 | Wandeln von Json nach Html | 30 |
| A.1 | Zeitmessung | 38 |
| B.1 | Docker-Compose | 41 |
| C.1 | Calling Script | 43 |
| C.2 | Aufrufe des Unterstützungsscriptes | 48 |
| D.1 | Performance Statistics Servlet | 49 |
| D.2 | Performance Statistics Provider | 49 |
| D.3 | Einbindung Servlet | 50 |
| E.1 | Vdi Logger | 51 |
| E.2 | Vdi Logger Factory | 52 |
| E.3 | Einbindung Factory | 52 |

ABKÜRZUNGSVERZEICHNIS

| | |
|------|-----------------------------------|
| EJB | Enterprise Java Beans |
| JSF | Java Server Faces |
| HTML | HyperText Markup Language |
| CSS | Cascading Style Sheets |
| SFSB | Stateful Session-Bean |
| JPA | Java Persistence API |
| JPQL | Java Persistence Query Language |
| API | Application Programming Interface |
| SQL | Structured Query Language |
| JVM | Java Virtual Machine |

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 wird 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. Ihre 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 Deutsche 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 Techniken 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 werden im Kapitel 3 die Konzepte vorgestellt, die die Stellen ermitteln, die eine schlechte Performance aufweisen und optimiert werden sollen. Hierzu gehören zum einen die Einstellungen der verwendeten Software, und zum anderen der Aufbau und die verwendeten Techniken in der Anwendung. Diese Techniken werden im weiteren Verlauf nochmal überprüft, ob eine alternative Lösung eine performantere Umsetzung bringen kann.

Bei den Performance-Untersuchung in Kapitel 4 werden nun die Konzepte angewandt, um die Umgebung selbst zu untersuchen und die dort bekannten Probleme zu ermitteln. Diese werden direkt bewertet, unter den Gesichtspunkten, ob eine Optimierung an dieser Stelle sinnvoll ist, oder ob der Arbeitsaufwand dafür zu enorm ist. *TODO: das enorm sollte man umschreiben* Zusätzlich werden noch die Vorbereitungen und die angepassten Konfigurationen für die nachfolgenden Performance-Untersuchung der Anwendung aufgezeigt.

Zuerst wird im Kapitel 5 die Ausgangsmessung durchgeführt, hierbei werden alle bekannten Caches deaktiviert und eine Messung durchgeführt. Dann werden Schicht für Schicht die Optimierungsmöglichkeiten aufgezeigt, umgesetzt und erneut gemessen. Diese Messung wird dann in Abhängigkeit zur Ausgangsmessung die Optimierung bewertet.

TODO: Kapitel 6 ist noch zu überlegen, ob das mit 7 nicht zusammengefasst werden könnte

Nach der Optimierung kommt nun die Evaluierung im Kapitel 6, um zu überprüfen ob die Anpassungen die gewünschten 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.

Es wird ab hier immer von einem *Glassfish*-Server geredet. In der Praxis wird ein *Payara*-Server verwendet. Der *Glassfish*-Server ist die Referenz-Implementierung von Oracle, welche für Entwickler bereitgestellt wird und die neuen Features unterstützt. Der *Payara*-Server ist aus dessen Quellcode entstanden und ist für Produktivumgebungen gedacht, da diese mit regelmäßigen Aktualisierungen versorgt wird. In dem weiteren Text wird weiterhin der Begriff *Glassfish* verwendet.

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 welchen *Controller* im Java Server Faces (JSF) die Anfrage weitergeleitet und verarbeitet wird. In diesem wird die Darstellung der Webseite geladen und die Anfragen für den darzustellenden Datenbestand abgeschickt.

Die Datenanfragen werden über die *EJB* an die *Java Persistence API (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.

TODO: Grafik anders positionieren

2.1 GLASSFISH - ENTERPRISE JAVA 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

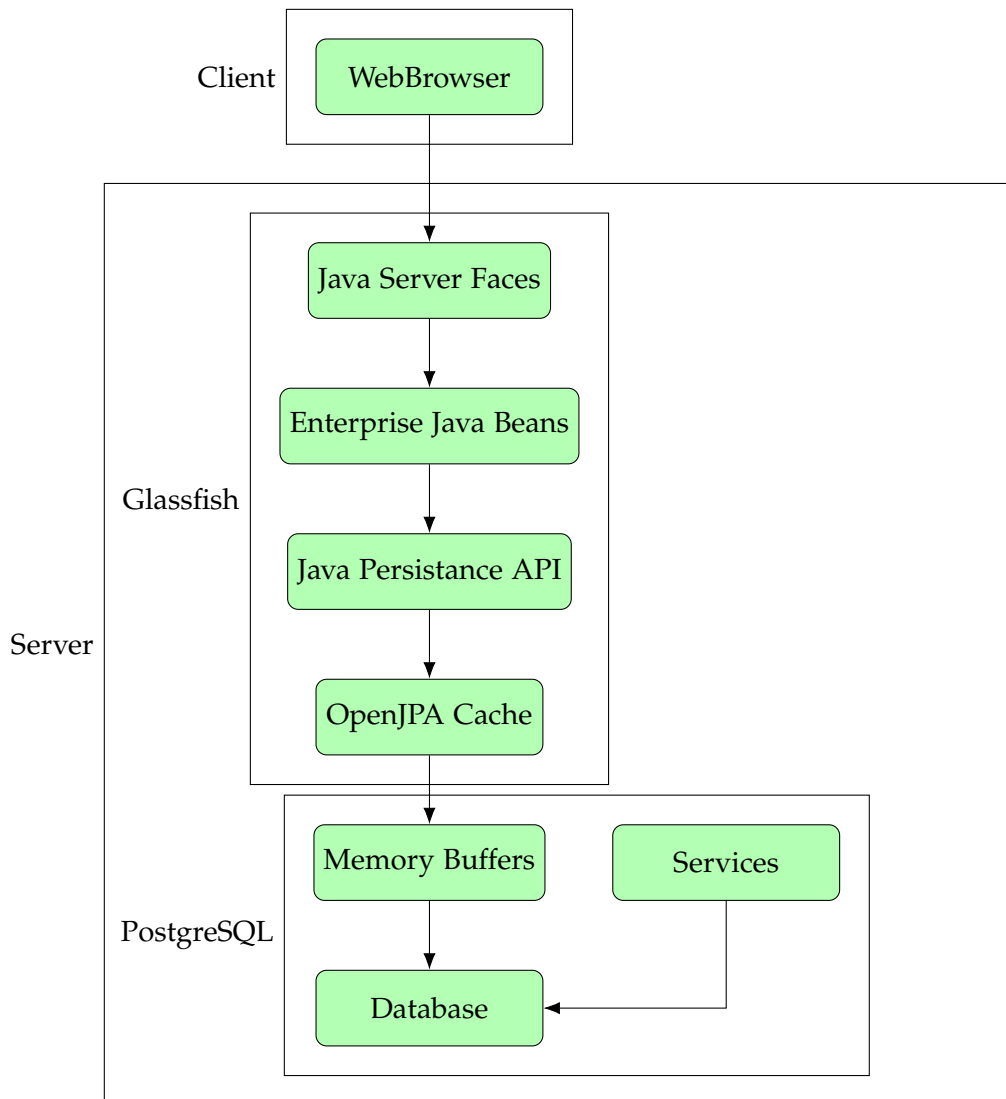


Abbildung 2.1: Ablauf einer Web-Anfrage

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.

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 [MW12, 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 *Veraltet* 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 [MW12, 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 beziehungsweise 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 circa 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 beziehungsweise 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. Ohne zu dem zusätzlichen Modul, können die Statistiken über die Software *pgBadger* erstellt werden. Dafür muss zusätzlich noch die Konfiguration des *PostgreSQL* angepasst 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 Produktivsysteme abgeglichen. Hierunter fallen die Einstellungen für die *shared_buffers*, der bei einem Arbeitsspeicher von mehr als 1 GB circa 25% des Arbeitsspeicher besitzen sollte [Posc].

Bei der Einstellung *temp_buffers* geht es um den Zwischenspeicher für jede Verbindung, die bei der Verwendung von temporären Tabellen verwendet wird. Dieser Wert sollte auf dem Standardwert von 8 MB belassen werden. Und nur bei der Verwendung von großen temporären Tabellen verändert werden.

Der Speicher, der für eine Abfrage verwendet werden darf, wird über die Konfiguration *work_mem* gesteuert. Wenn der Speicher zu gering wird, werden die Zwischenergebnisse in temporäre Dateien ausgelagert. Der empfohlene Wert berechnet sich aus *shared_buffers* dividiert durch *max_connections* [Con]. Sollte der Berechnung außerhalb der Grenzwerte von 1 MB und 256 MB liegen, ist der jeweilige Grenzwert zu verwenden. Um zu ermitteln, ob die Konfiguration richtig ist, muss im PostgreSQL die Einstellung *log_temp_files* auf 0 gesetzt werden. Mit dieser kann ermittelt, ob temporäre Dateien verwendet werden und deren Größe. Bei vielen kleineren Dateien sollte der Grenzwert erhöht werden. Bei wenigen großen Dateien ist es sinnvoll den Wert so zu belassen.

Für die Wartungsaufgaben wie VACUUM oder dem erstellen von Indizes wird die Begrenzung über die Einstellung *maintenance_work_mem* gesetzt. Dieser Wert sollte 5% des verfügbaren Arbeitsspeicher entsprechen und größer als *work_mem* sein.

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 Informationen 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 3.1 gezeigt. Zusätzlich wird für jedes dargestellte Dokument eine zusätzliche Abfrage durchgeführt, die in 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_birch_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_birch_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 circa 5400 Briefe, für die jeweils zwei bis sieben eingescannte Faksimile gespeichert werden. Diese Graphik-Dateien werden im TIFF-Format abgespeichert und benötigen zwi-

schen 1 und 80 MB Speicherplatz. Dadurch kommt die Datenbank aktuell auf circa 3,8 GB.

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.

TODO: Statistik-Webseite hier entfernen, wird aktuell nicht angewendet

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 werden die Webseite über ein Script aufgerufen und die Aufrufzeiten sowie andere externe Statistiken darüber erstellt und gespeichert.

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. Hier werden nur die externen Statistiken durch das Skript verwendet, um zu prüfen in welchen Umfang die Umstellungen eine Veränderung im Verhalten der Webseite bewirken.

Bei den JSF wird eine Zeitmessung eingefügt. Hierfür wird eine *Factory* eingebaut, die sich in die Verarbeitung der Seiten einhängt, und damit die Zeiten für das Ermitteln der Daten, das Zusammensetzen und das Render der Sicht aufgenommen werden können. Die Zeiten werden in die Log-Datei des *Glassfish*-Servers hinterlegt und durch das Skript ausgewertet. Somit ist es einfach aufzuzeigen, an welcher Stelle der größte Teil der Verzögerung auftritt.

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 ist. Neben der Abfrage der Daten beinhaltet diese auch noch 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.

Damit die Messungen nachvollziehbar bleiben, werden die Testaufrufe durch ein Bash-Script automatisiert gerufen. Wichtig hierbei ist, dass die Webseite immer vollständig gerendert vom Server an den Client übertragen wird. Somit kann die clientseitige Performance ignoriert werden, da alle Daten direkt in dem einen Aufruf bereitgestellt wird. In dem Skript werden zum einen die Laufzeiten der Webanfragen ermittelt und die kürzeste, die längste und die durchschnittliche Laufzeit ermittelt. Aufgrund der Speicherprobleme, werden auch die Speicherbenutzung des *Glassfish*-Servers vor und nach den Aufrufen ermittelt. Zum Schluss werden noch die Log-Dateien des *PostgreSQL*-Servers über das Tool *pgBadger* analysiert und als Bericht aufbereitet.

Um die Netzwerklatenz ignorieren zu können, wird das Skript auf dem gleichen Computer aufgerufen, auf dem der Webserver gestartet wurde. Das zugehörige Script ist im Anhang A zu finden.

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

Die einfachste Art die Einstellungen zu prüfen ist, die Abfrage in 4.1 am Datenbankserver auszuführen.

Listing 4.1: Ermitteln der PostgreSQL Einstellungen

```
SELECT name      AS setting_name
       , setting AS setting_value
       , unit     AS setting_unit
FROM   pg_settings
WHERE  name IN (
        'shared_buffers'
      , 'temp_buffers'
      , 'work_mem'
      , 'max_connections'
      , 'maintenance_work_mem'
      )
```

TODO: Konfiguration vom Produktionsserver prüfen

4.4 EINBAU UND AKTIVIEREN VON PERFORMANCE-MESSUNG

Um eine Messung der Performance in der Webseite durchführen zu können, gibt es in JSF die Möglichkeit, über eine eigene Implementierung der Klasse **ViewDeclarationLanguageWrapper** sich in das generieren der Webseite einzuhängen. Hierbei können die Funktionen für das Erstellen, des Bauen und das Rendern der Webseite überschrieben werden. In den überschriebenen Funktionen werden nun Laufzeiten gemessen und die ermittelten Zeiten mit einer Kennung in die Log-Datei eingetragen. Durch die Kennung, können die Zeiten im Nachgang über ein Script ermittelt und ausgewertet werden.

Zusätzlich wird noch eine Implementierung der zugehörigen Factory-Klasse **ViewDeclarationLanguageFactory** benötigt. Durch diese Factory-Klasse wird der eigentlichen Wrapper mit der Performance-Messung in die Bearbeitungsschicht eingehängt. Diese Implementierung wird dann noch in der **faces-config.xml** eingetragen, wie das in 4.2 gezeigt wird, damit die Factory durch das System aufgerufen wird.

Listing 4.2: Einbindung Factory

```
<factory>
  <view-declaration-language-factory>
    de.wedekind.utils.VdlLoggerFactory
  </view-declaration-language-factory>
</factory>
```

Der Quellcode der Klassen ist im Anhang E zu finden.

Um die Abfragen im *PostgreSQL* untersuchen zu können, ist es am leichtesten, wenn man die Konfiguration so anpasst, dass alle Abfragen mit entsprechenden Zeitmessungen in die Log-Datei des ausgegeben werden. Zuerst werden über die Einstellungen unter 4.3 die Datei und das Format definiert.

Listing 4.3: PostgreSQL Dateikonfiguration

```
log_destination = 'jsonlog'
logging_collector = on
log_directory = 'log'
log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'
log_file_mode = 0640
log_rotation_size = 100MB
```

Über die Konfiguration unter 4.4 wird definiert welche Werte mit protokolliert werden. Die wichtigste Einstellung ist *log_min_duration_statement*, diese definiert ab welcher Laufzeit eine Abfrage protokolliert werden soll. Mit dem Wert 0 werden alle Abfragen protokolliert. Alle weitere Einstellungen sind so gesetzt, dass nicht unnötige Abfragen für die spätere Auswertung mit *pgBadger* protokolliert werden. Zusätzlich ist die Einstellung *log_temp_files* auf 0 zu setzen. Dadurch werden alle erzeugten temporären Dateien und ihre Größe ebenfalls protokolliert. Diese Dateien entstehen, wenn der temporäre Puffer für die Abfrage nicht ausreicht und die Zwischenergebnisse ausgelagert werden müssen.

Listing 4.4: PostgreSQL Ausgabekonfiguration

```
log_min_duration_statement = 0
log_autovacuum_min_duration = 10
log_checkpoints = off
log_connections = on
log_disconnections = on
log_disconnections = on
log_duration = off
```

```
log_error_verbosity = default
log_hostname = on
log_lock_waits = on
log_statement = 'none'
log_temp_files = 0
log_timezone = 'Europe/Berlin'
```

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

Für die Tests wird ein aktuelles Manjaro-System mit frisch installierten Payara als Serverhost und der IntelliJ IDEA als Entwicklungsumgebung verwendet. Der Computer ist mit einer Intel CPU i7-12700K, 32 GB Arbeitsspeicher und einer SSD als Systemfestplatte ausgestattet.

Zur ersten Untersuchung und der Bestimmung der Basis-Linie, wurde das Script ohne eine Änderung an dem Code und der Konfiguration mehrfach aufgerufen. Hierbei hat sich gezeigt, dass der erste Aufruf nach dem Deployment circa 1500 ms gedauert hat. Die weiteren Aufrufe benötigen im Durchschnitt noch 600 ms. Beim achten Aufruf des Scripts hat der Server nicht mehr reagiert und im Log ist ein `OutOfMemoryError` protokolliert worden.

Nach einem Neustart des Servers, konnte das gleiche Verhalten wieder reproduziert werden. Daraufhin wurde das Test-Script um die Anzeige der aktuellen Speicherverwendung des Payara-Servers erweitert und diese zeitgleich zu beobachten. Diese Auswertung zeigte, dass der Server mit circa 1500 MB RSS Nutzung an seine Grenzen stößt. Diese Grenzen wurde durch die Konfigurationsänderung im Payara-Server von `-Xmx512m` auf `-Xmx4096m` nach oben verschoben. Nun werden circa 60 Aufrufe des Scripts benötigt, damit der Server nicht mehr reagiert. Hierbei wird aber kein `OutOfMemoryError` in der Log-Datei protokolliert und der Server verwendet nun circa 4700 MB RSS. Bei allen Tests war noch mehr als die Hälfte des verfügbaren Arbeitsspeichers des Computers ungenutzt.

Mit der Konfiguration `-Xmx` wird der maximal verwendbare Heap-Speicher in der Java Virtual Machine (JVM) definiert. Dies zeigt direkt, dass es ein Problem in der Freigabe der Objekte gibt, da dass Erhöhen des verwendbaren Arbeitsspeichers das Problem nicht löst, sondern nur verschiebt.

Für alle nachfolgenden Messungen wird das Skript C verwendet, welches die einzelnen Aufrufe steuert. Die Ergebnisse werden in eine Tabelle überführt, wie in der Tabelle 5.1. Hierbei werden die Aufrufzeiten der Webseite aus dem Skript für die Zeitmessung mit Mindest-, Durchschnitt- und Maximalzeit aufgenommen, hierbei ist eine kürzere Zeit besser. Zusätzlich wird die Anzahl der aufgerufenen SQL Abfragen ermittelt, auch hier gilt, dass weniger Aufrufe besser sind. Als letztes wird noch der verwendete Arbeitsspeicher vom *Glassfish*-Server vor und nach dem Aufruf ermittelt und die Differenz gebildet, hierbei sollte im besten Fall die Differenz bei 0 liegen. Dieser Aufbau gilt für alle weiteren Messungen. Zusätzlich werden noch die Laufzeiten der JSF ermittelt und die durchschnittlichen Zeiten mit in der Tabelle dargestellt, und auch hier ist es besser, wenn die Zeiten kürzer sind.

Als Grundlage für die Vergleiche wurden eine Messung durchgeführt, bei der alle Caches deaktiviert wurden und keine Änderung am Code vorgenommen wurde. Das Ergebnis dieser Messung ist in 5.1 zu finden. Diese zeigen auch direkt ein erwartetes Ergebnis, dass der erste Aufruf bedeutend länger dauert als die Nachfolgenden. Ebenfalls sieht man eindeutig, dass die Anzahl der Anfragen nach dem ersten Aufruf immer die gleiche Anzahl besitzen. Der Speicherbedarf steigt auch relative gleichmässig, was nicht recht ins Bild passt, da hier keine Objekte im Cache gehalten werden sollten.

TODO: Diese Tabelle vielleicht auch einfach komplett streichen, da der Datenbestand anders ist, und das wichtigste die Zeit der SQL-Abfragen nicht sichtbar ist

| # | Aufrufzeit (ms) | | | | RSS (MB) | | |
|---|-----------------|-----|------|---------|----------|---------|--------|
| | min | avg | max | Queries | davor | danach | diff |
| 1 | 395 | 578 | 1312 | 12237 | 747.15 | 924.88 | 177.73 |
| 2 | 353 | 375 | 464 | 12080 | 924.51 | 1027.75 | 103,24 |
| 3 | 286 | 345 | 535 | 12080 | 1018.21 | 1145.36 | 127.15 |
| 4 | 291 | 307 | 340 | 12080 | 1129.91 | 1239.75 | 109,84 |

Tabelle 5.1: Messung ohne Caches

Vor jedem weiteren Test-Lauf wurde die Domain beendet und komplett neugestartet, um mit einer frischen Instanz zu beginnen. Hierbei ist aufgefallen, dass fast immer 62 Abfragen zur Startup-Phase dazugehört haben, unabhängig von den konfigurierten Cache Einstellungen. Einige dieser Abfragen sind durch das Erstellen der Materialisierten Sichten **searchreference** und *searchfulltext* erklärbar. Zusätzlich ist noch ein zyklischer Dienst *SearchEntityService* vorhanden, der zum Start und alle sechs Stunden den Datenbestand für die Suche aufbereitet und entsprechend einige Abfragen an die Datenbank absetzt. Da weder die Sichten noch der Dienst für die Dokumentenliste benötigt werden, wurde der Dienst und das Erstellen im Code für die weiteren Tests deaktiviert.

Da die Abfragezeiten auf der Datenbank zu gering waren, um eine Verbesserung feststellen zu können, wurde für den PostgreSQL und den Payara-Server ein Docker-Container erzeugt und diese limitiert. Die Konfiguration ist im Anhang B beschrieben.

Mit dem neuen Aufbau ergeben sich nun neue Messungen. Für den Speicherbedarf wird nun nicht mehr der benutzte Speicher der Anwendung beobachtet, sondern die Speichernutzung des Docker-Containers für den Payara-Server. Auch hier ist es besser, wenn es keine oder nur geringe Änderungen vor und nach dem Aufruf der Webseite gibt, ein steigender Wert zeigt an, dass der verwendete Speicher nicht sauber freigegeben werden kann.

Für die Ausführungszeiten der SQL-Abfragen wurden nur die sechs Abfragen für die Darstellung der Tabelle beachtet. Hierzu zählt die Hauptabfrage der Dokumenten--Tabelle, die Ermittlung des letzten und ersten Eintrags in der Tabelle, die Ermittlung der Adressen des Autors, die Ermittlung der

Koautoren, die Ermittlung der Faksimile, sowie die Ermittlung der Anzahl aller vorhandenen Dokumente.

Zusätzlich wird die Zeit des Rendern der Sicht gemessen. Hierbei wird zum einen die komplette Zeit des Renderns ermittelt. Innerhalb des Rendern wird dann noch die Zeit gemessen, wie lange es benötigt, die Daten aus der Datenbank zu laden, und in die Java-Objekte umzuformen.

| # | Aufrufzeit (ms) | | | Queries (ms) | | Memory (MB) | | | Render (ms) | | | DB-load (ms) | | |
|---|-----------------|-----|------|--------------|------|-------------|--------|-------|-------------|-----|------|--------------|-----|-----|
| | min | avg | max | #-avg | avg | start | stop | diff | min | avg | max | min | avg | max |
| 1 | 451 | 682 | 1931 | 1223.0 | 30.3 | 931.3 | 986.1 | 54.8 | 440 | 666 | 1859 | 290 | 399 | 710 |
| 2 | 341 | 389 | 478 | 1208.0 | 31.2 | 986.5 | 1159.0 | 172.5 | 331 | 378 | 468 | 235 | 282 | 367 |
| 3 | 299 | 407 | 682 | 1208.0 | 33.5 | 1163.0 | 1273.0 | 110.0 | 290 | 398 | 672 | 207 | 307 | 579 |
| 4 | 278 | 359 | 424 | 1208.0 | 33.7 | 1272.0 | 1465.0 | 193.0 | 270 | 351 | 415 | 198 | 269 | 342 |
| 5 | 264 | 317 | 356 | 1208.0 | 32.9 | 1467.0 | 1574.0 | 107.0 | 256 | 309 | 348 | 184 | 235 | 276 |

Tabelle 5.2: Messung ohne Caches im Docker

5.1 UMGESTALTEN DER DATENBANKTABELLEN

Hierfür wurde die aktuelle Datenstruktur untersucht um zu prüfen, ob eine Umgestaltung der Tabelle einen Verbesserung bringen würden. Die typische Optimierung ist die Normalisierung der Tabellenstruktur. Die Tabellenstruktur ist aktuell schon normalisiert, daher kann hier nichts weiter optimiert werden.

Eine weitere Optimierungsstrategie besteht in der Denormalisierung, um sich die Verknüpfungen der Tabellen zu sparen. Dies ist in diesem Fall nicht anwendbar, da nicht nur 1:n Beziehungen vorhanden sind, sondern auch auch n:m Beziehungen. Dadurch würden sich die Anzahl der Dokumentenliste erhöhen. Oder man muss die Duplikate auf der Serverseite zusammenführen. *TODO: Gefällt der Verena nicht so*

5.2 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. Bei der Berechnung der Anzahl der Element werden angeheftete Objekte nicht beachtet.

Die Anzahl der Soft References kann ebenfalls über eine Einstellung gesteuert werden. Hierfür wird die Anzahl der Elemente über `SoftReferenceSize` gesetzt, dessen Wert im Standard auf `unbegrenzt` steht. Mit dem Wert `0` werden die Soft References komplett deaktiviert. Über die Attribute an den Entitätsklassen, können diese Referenzen ebenfalls gesteuert werden, hierzu muss eine Überwachungszeit angegeben werden. Diese Zeit gibt in ms

an, wie lange ein Objekt gültig bleibt. Mit dem Wert -1 wird das Objekt nie ungültig, was ebenfalls der Standardwert ist.

Zuerst wird mit aktivierten Cache mit einer Cache-Größe von 1000 Elemente getestet. Wie in 5.3 zu sehen, dauert auch hier der erste Aufruf minimal länger als ohne aktiviertem Cache. Alle Nachfolgenden Aufrufe wiederum sind um 100ms schneller in der Verarbeitung. Auch bei der Anzahl der Anfragen an die Datenbank kann der Rückgang der Anfragen sehr gut gesehen werden. Aktuell kann die Verringerung des wachsenden Speicherbedarfs nur nicht erklärt werden.

| # | Aufrufzeit (ms) | | | Queries (ms) | | Memory (MB) | | | Render (ms) | | | DB-load (ms) | | |
|---|-----------------|-----|------|--------------|------|-------------|--------|-------|-------------|-----|------|--------------|-----|-----|
| | min | avg | max | #-avg | avg | start | stop | diff | min | avg | max | min | avg | max |
| 1 | 291 | 611 | 2347 | 730.2 | 28.8 | 852.7 | 891.9 | 39.2 | 282 | 595 | 2286 | 172 | 284 | 770 |
| 2 | 278 | 319 | 422 | 667.3 | 25.8 | 892.7 | 1010.0 | 117.3 | 266 | 309 | 411 | 173 | 195 | 220 |
| 3 | 229 | 281 | 329 | 680.6 | 27.6 | 1011.0 | 1067.0 | 56.0 | 220 | 271 | 313 | 134 | 180 | 222 |
| 4 | 222 | 280 | 321 | 671.3 | 27.6 | 1067.0 | 1122.0 | 55.0 | 213 | 271 | 310 | 131 | 189 | 238 |
| 5 | 206 | 272 | 388 | 683.6 | 27.6 | 1122.0 | 1219.0 | 97.0 | 199 | 264 | 380 | 122 | 175 | 291 |

Tabelle 5.3: Messung mit OpenJPA-Cache und Größe auf 1000

Bei einer erhöhten Cache-Größe, von 1000 auf 10000, zeigt sich auf den ersten Blick ein noch besseres Bild ab, wie in 5.4 ersichtlich ist. Der erste Aufruf entspricht der Laufzeit mit geringerer Cache-Größe, aber schon die Anfragen an die Datenbank gehen drastisch zurück. Bei den weiteren Aufrufen werden im Schnitt nun nur noch 6 Anfragen pro Seitenaufruf an die Datenbank gestellt, wodurch die Laufzeit im Schnitt nochmal um 100 ms beschleunigt werden konnte.

| # | Aufrufzeit (ms) | | | Queries (ms) | | Memory (MB) | | | Render (ms) | | | DB-load (ms) | | |
|---|-----------------|-----|------|--------------|------|-------------|-------|------|-------------|-----|------|--------------|-----|-----|
| | min | avg | max | #-avg | avg | start | stop | diff | min | avg | max | min | avg | max |
| 1 | 151 | 368 | 1904 | 141.2 | 20.8 | 906.3 | 936.8 | 30.5 | 164 | 404 | 2232 | 39 | 124 | 847 |
| 2 | 133 | 143 | 159 | 6.0 | 20.5 | 935.7 | 939.3 | 3.6 | 121 | 136 | 146 | 32 | 36 | 44 |
| 3 | 120 | 126 | 132 | 6.0 | 19.9 | 939.4 | 942.7 | 3.3 | 116 | 136 | 256 | 32 | 47 | 167 |
| 4 | 120 | 124 | 128 | 6.0 | 21.4 | 944.3 | 945.4 | 1.1 | 105 | 113 | 125 | 30 | 32 | 39 |
| 5 | 109 | 114 | 131 | 6.0 | 19.7 | 945.5 | 946.7 | 1.2 | 101 | 107 | 112 | 30 | 32 | 35 |

Tabelle 5.4: Messung mit OpenJPA-Cache und Größe auf 10000

Bei dem deaktivieren der *SoftReference* und dem kleineren Cache zeigt sich keine große Differenz. Somit scheint die *SoftReference* nicht das Problem für den steigenden Arbeitsspeicher zu sein, wie in Tabelle 5.5 zu sehen.

| # | Aufrufzeit (ms) | | | Queries (ms) | | Memory (MB) | | | Render (ms) | | | DB-load (ms) | | |
|---|-----------------|-----|------|--------------|------|-------------|--------|-------|-------------|-----|------|--------------|-----|-----|
| | min | avg | max | #-avg | avg | start | stop | diff | min | avg | max | min | avg | max |
| 1 | 339 | 659 | 2435 | 880.8 | 33.2 | 909.6 | 960.2 | 50.6 | 330 | 644 | 2375 | 218 | 343 | 815 |
| 2 | 267 | 332 | 388 | 832.1 | 28.1 | 959.7 | 1000.0 | 40.3 | 259 | 323 | 377 | 178 | 229 | 280 |
| 3 | 265 | 397 | 350 | 830.3 | 27.3 | 1001.0 | 1107.0 | 106.0 | 256 | 288 | 241 | 172 | 204 | 252 |
| 4 | 249 | 311 | 401 | 727.8 | 27.1 | 1108.0 | 1234.0 | 126.0 | 240 | 303 | 392 | 165 | 225 | 317 |
| 5 | 268 | 296 | 325 | 931.9 | 28.0 | 1236.0 | 1239.0 | 3.0 | 260 | 288 | 318 | 192 | 217 | 244 |

Tabelle 5.5: Messung mit OpenJPA-Cache und Größe auf 1000 und 0 *SoftReference*

Der Vergleich zeigt, dass der Cache eine gute Optimierung bringt, aber dies nur dann gut funktioniert, wenn immer wieder die gleichen Objekte ermittelt werden. Sobald die Anfragen im Wechsel gerufen werden oder einfach nur die Menge der Objekte den Cache übersteigt, fällt die Verbesserung geringer aus.

5.3 CACHED QUERIES

Über die Einstellung `openjpa.jdbc.QuerySQLCache` wird der Cache für abfragen aktiviert. Hierbei können Abfragen angegeben werden, die aus dem Cache ausgeschlossen werden. Der QueryCache wiederum beachtet aber nur Abfragen die keine Parameter verwenden. Das sieht man auch entsprechend der Auswertung der Aufrufe 5.6, dass hier keine Veränderung der Aufrufzeiten stattgefunden hat. Gleich ob man mit Java Persistence Query Language (JPQL) oder mit der Criteria API abfragt.

| # | Aufrufzeit (ms) | | | Queries (ms) | | Memory (MB) | | | Render (ms) | | | DB-load (ms) | | |
|---|-----------------|-----|------|--------------|-----|-------------|--------|-------|-------------|-----|------|--------------|-----|-----|
| | min | avg | max | #-avg | avg | start | stop | diff | min | avg | max | min | avg | max |
| 1 | 409 | 771 | 2660 | 1222.4 | xxx | 850.4 | 982.8 | 132.4 | 366 | 633 | 2019 | 254 | 364 | 758 |
| 2 | 336 | 387 | 504 | 1208.0 | xxx | 982.9 | 1113.0 | 130.1 | 310 | 374 | 433 | 221 | 268 | 345 |
| 3 | 312 | 373 | 422 | 1208.0 | xxx | 1114.0 | 1221.0 | 107.0 | 295 | 401 | 658 | 216 | 320 | 570 |
| 4 | 288 | 363 | 471 | 1208.0 | xxx | 1239.0 | 1474.0 | 235.0 | 269 | 356 | 486 | 200 | 279 | 405 |
| 5 | 325 | 398 | 535 | 1208.0 | xxx | 1474.0 | 1666.0 | 192.0 | 280 | 466 | 804 | 208 | 390 | 725 |

Tabelle 5.6: Messung mit aktiviertem Cached Queries

5.4 CACHING IM JPA

TODO: muss noch umgebaut werden, falsche Konfiguration erwischt

5.5 CACHING IN EJB

Die Cache-Einstellungen des EJB sind in der Admin-Oberfläche des Payara-Servers zu erreichen. Hier *TODO: Cache config noch definieren*

| # | Aufrufzeit (ms) | | | Queries (ms) | | Memory (MB) | | | Render (ms) | | | DB-load (ms) | | |
|---|-----------------|-----|------|--------------|---------|-------------|--------|------|-------------|-----|-----|--------------|-----|-----|
| | min | avg | max | #-avg | avg | start | stop | diff | min | avg | max | min | avg | max |
| 1 | 416 | 554 | 1269 | 12237 | 840.31 | 998.07 | 157.76 | | | | | | | |
| 2 | 299 | 394 | 749 | 12080 | 973.20 | 1101.37 | 128.17 | | | | | | | |
| 3 | 293 | 324 | 382 | 12080 | 1092.00 | 1192.87 | 100.87 | | | | | | | |
| 4 | 281 | 318 | 398 | 12080 | 1191.25 | 1305.29 | 114.04 | | | | | | | |

Tabelle 5.7: Messung mit EJB-Cache

5.6 ABFRAGEN JPQL

Für die JPQL wird ein Structured Query Language (SQL) ähnlicher Syntax verwendet um die Abfragen an die Datenbank durchzuführen. Für die Dokumentenliste wird der Code aus 5.1 verwendet. Die Namen mit vorangestellten Doppelpunkt sind Übergabevariablen.

Listing 5.1: JPQL Dokumentenliste

```
SELECT DISTINCT d FROM Document d
LEFT JOIN FETCH d.authorPerson
LEFT JOIN FETCH d.coauthorPersonSet
LEFT JOIN FETCH d.addresseePersonSet
WHERE d.validUntil > :now
AND d.isPublishedInDb = :published
ORDER BY d.documentId ASC
```

In dem dazugehörigen Code am Server wird der JPQL-Code als Name-`dQuery` hinterlegt und über den Name `Document.findAll` referenziert. Eine Veränderung der Abfrage ist hier leider nicht möglich, wie man im Code 5.2 sehen kann.

Listing 5.2: Java JPQL Dokumentenliste

```
List<Document> myResultList = createNamedTypedQuery("Document.findAll")
    .setParameter("now", _IncludeDeleted ? new Date(0) : Date.from(
        LocalDateTime.now().atZone(ZoneId.systemDefault()).toInstant()))
    .setParameter("published", true)
    .setFirstResult(_Start)
    .setMaxResults(_Size)
    .setHint("javax.persistence.query.fetchSize", _Size)
    .getResultList();

// Uebergabe der Ergebnisliste
if(myResultList != null && !myResultList.isEmpty()) {
    myResult.addAll(myResultList);
}
```

Da dieser Code direkt so aus dem Projekt kommt, wird hierfür keine gesonderte Zeitmessung durchgeführt, da diese der Messung 5.1 entspricht.

Für die Optimierung wurden noch zusätzlich die Hints `openjpa.hint.OptimizeResultCount`, `javax.persistence.query.fetchSize` und `openjpa.FetchPlan.FetchBatchSize` gesetzt. Hierbei konnten je nach gesetztem Wert, keine relevanten Unterschiede festgestellt werden. Hierbei wurde der Wert auf zwei gesetzt, welcher viel zu gering ist. Als weiterer Test wurde der Wert auf angefragte Größe gestellt und auf den 20-fachen Wert der angefragten Größe.

Ebenso bringt der Hint `openjpa.FetchPlan.ReadLockMode` auch keinen Unterschied bei der Geschwindigkeit. Hierbei ist erklärbar, da im Standard bei einer reinen Selektion eine Lesesperre aktiv sein muss. Bei `openjpa.FetchPlan.Isolation` wird gesteuert, auf welche Sperren beim laden geachtet wird. Damit könnte man zwar Schreibsperrern umgehen, und würde damit die Anfrage nicht

mehr blockieren lassen, aber es führt unweigerlich zu sogenannten „Dirty-Reads“, wodurch die Ausgabe verfälscht werden könnte. Daher ist diese Einstellung sehr mit Vorsicht zu verwenden.

Mit dem Hint `openjpa.FetchPlan.EagerFetchMode` wird definiert, wie zusammengehörige Objekte abgefragt werden. Bei dem Wert `none` werden nur die Basis-Daten abgefragt und jedes weitere Objekt wird in einem eigenen Statement abgefragt. Mit `join` wird definiert, dass abhängige Objekte die als „to-one“-Relation definiert sind, in der Abfrage über einen Join verknüpft und damit direkt mitgeladen werden. Bei reinen „to-one“-Relation funktioniert das rekursive und spart sich damit einige einzelne Abfragen. Bei der Einstellung `parallel` wird für zwar für jedes abhängigen Objektdefinition eine Abfrage durchgeführt, aber bei dieser wird der Einstieg über das Hauptobjekt durchgeführt. Somit muss in unserem Beispiel nicht für jedes Dokument eine einzelne abfrage für die CoAutoren durchgeführt werden, sondern es wird nur eine Abfrage abgesetzt für alle Dokumente die ermittelt wurden. Technisch gesehen wird, die gleiche WHERE-Abfrage nochmal durchgeführt und um die JOINS ergänzt, um die Daten der Unterobjekte zu ermitteln. Mit dem Hint `openjpa.FetchPlan.SubclassFetchMode` ist die Konfiguration für Unterklassen definiert. Die Möglichkeiten entsprechen der vom `openjpa.FetchPlan.EagerFetchMode`.

Beim Umstellen der 2 Hints auf `parallel` wird die Bearbeitungszeit fast halbiert und Anzahl der Datenbankaufrufe wurde fast geviertelt. Dies zeigt, dass die einzelnen Aufrufe je Dokument aufwendiger sind, als eine komplette Abfrage der abhängigen Daten und das zusammensetzen in der OpenJPA-Schicht.

Der letzte Hint `openjpa.FetchPlan.MaxFetchDepth` schränkt die rekursive Tiefe ein, für die abhängige Objekte mitgeladen werden. Damit wird zwar die Abfrage beschleunigt, aber nur aufgrund fehlender Datenbestände. **TODO: besser formulieren**

5.7 ABFRAGEN CRITERIA API

Für die Criteria API wird die Abfrage nicht in einem SQL-Dialekt beschreiben. Hierbei werden über Attribute die Verlinkung zur Datenbank durchgeführt. An der Klasse selbst wird der Tabellename definiert und an den Attributen die Spaltennamen. Um die Anfrage durchführen muss nun noch Datenklasse angegeben werden und mit den Parametern versorgt werden, wie es in 5.3 gezeigt wird.

Listing 5.3: Criteria API Dokumentenliste

```
CriteriaBuilder cb = getEntityManager().getCriteriaBuilder();
CriteriaQuery<Document> cq = cb.createQuery(Document.class);
Root<Document> from = cq.from(Document.class);
ParameterExpression<Boolean> includedPara = cb.parameter(Boolean.class,
    "published");
ParameterExpression<Date> validPart = cb.parameter(Date.class, "now");

CriteriaQuery<Document> select = cq.select(from)
```

```

        .where(cb.and(
            cb.equal(from.get("isPublishedInDb"), includedPara),
            cb.greaterThan(from.get("validUntil"), validPart)
        ));
TypedQuery<Document> typedQuery = getEntityManager().createQuery(select)
    .setParameter("now", _IncludeDeleted ? new Date(0) : Date.from(
        LocalDateTime.now().atZone(ZoneId.systemDefault()).toInstant()))
    .setParameter("published", true)
    .setFirstResult(_Start)
    .setMaxResults(_Size)
    .setHint("javax.persistence.query.fetchSize", _Size);
List<Document> myResultList = typedQuery.getResultList();

// Uebergabe der Ergebnisliste
if (myResultList != null && !myResultList.isEmpty()) {
    myResult.addAll(myResultList);
}

```

Wie in der Messung 5.8 zu sehen, unterscheiden sich die Abfragezeiten nur marginal von denen mit JPQL. Wenn man sich den Code im Debugger anschaut, sieht man auch, dass die zusammengesetzten Abfragen in den Java-Objekten fast identisch sind. Und in der Datenbank sind die Anfragen identisch zu denen über JPQL.

| # | Aufrufzeit (ms) | | | Queries (ms) | | Memory (MB) | | | Render (ms) | | | DB-load (ms) | | |
|---|-----------------|-----|------|--------------|---------|-------------|--------|------|-------------|-----|-----|--------------|-----|-----|
| | min | avg | max | #-avg | avg | start | stop | diff | min | avg | max | min | avg | max |
| 1 | 396 | 572 | 1535 | 12173 | 796.59 | 970.10 | 173.51 | | | | | | | |
| 2 | 333 | 366 | 397 | 12080 | 982.28 | 1064.12 | 81.84 | | | | | | | |
| 3 | 286 | 339 | 554 | 12080 | 1048.12 | 1162.92 | 114.80 | | | | | | | |
| 4 | 293 | 317 | 388 | 12080 | 1150.43 | 1263.77 | 113.34 | | | | | | | |

Tabelle 5.8: Messung mit Criteria-API ohne Cache

Daher bringt die Criteria API keinen performance Vorteil gegenüber der JPQL-Implementierung. Somit können beide Implementierung ohne bedenken gegeneinander ausgetauscht werden, und die verwendet werden, die für den Anwendungsfall einfacher umzusetzen ist.

Bei den Hints ist es das gleiche wie bei JPQL. Auch hier haben die meisten Hints keinen merkbaren Einfluss. Die Einstellung `openjpa.FetchPlan.EagerFetchMode` liefert auch hier Optimierungen, wenn der Wert auf `parallel` gestellt wird. Hier wird ebenfalls die Anzahl der Anfragen reduziert und damit auch die Geschwindigkeit optimiert.

5.8 MATERIALIZED VIEWS

Materialized Views sind Sichten in der Datenbank, die beim erstellen der Sicht den aktuellen Zustand ermitteln und Zwischenspeichern. Somit wird beim Zugriff auf diese Sichten, nicht die hinterlegte Abfrage ausgeführt, sondern auf die gespeicherten Daten zugegriffen. Dies ist gerade bei vielen Joins

von Vorteil. Zusätzlich können auf solchen Sichten auch Indexe erstellt werden, um noch effektiver die Abfragen bearbeiten zu können.

Der größte Nachteil dieser Sichten ist, dass sie zyklisch oder bei Datenänderungen aktualisiert werden müssen, sonst läuft der Datenbestand der Sicht und der zugrundeliegenden Abfrage auseinander. Da die Hauptarbeiten auf der Webseite die Abfrage der Daten ist, und nicht das editieren, kann dieser Nachteil bei entsprechender Optimierung ignoriert werden.

In diesem Test, wurde zusätzlich zur normalen Abfragen noch die nachfolgenden einzelabfragen als Sub-Selects hinzugefügt, wie in 5.4 zu sehen. Somit können die nachfolgenden einzelnen Abfragen eingespart werden. Dies wiederum geht aber auf die Performance der Erstellung der Sicht und ihrer Aktualisierung.

Listing 5.4: SQL Materialized View

```

CREATE MATERIALIZED VIEW searchdocument AS
SELECT
  d.id, d.documentId, d.datatype, d.startdatestatus, d.startyear,
  d.startmonth, d.startday, d.enddatestatus, d.endyear, d.endmonth,
  d.endday,
  (
    SELECT
      jsonb_build_object(
        'personId', hp.personid,
        'surname', hp.surname,
        'firstname', hp.firstname,
        'dateBirth', json_build_object(
          'year', hp.birthstartyear,
          'month', hp.birthstartmonth,
          'day', hp.birthstartday
        ),
        'dateDeath', json_build_object(
          'year', hp.deathstartyear,
          'month', hp.deathstartmonth,
          'day', hp.deathstartday
        )
      )
    FROM historicalperson hp
    WHERE hp.id = d.authorperson_id
    AND hp.validuntil > NOW()
  ) as author,
  (
    SELECT
      jsonb_agg(jsonb_build_object(
        'personId', hcap.personid,
        'surname', hcap.surname,
        'firstname', hcap.firstname,
        'dateBirth', json_build_object(
          'year', hcap.birthstartyear,
          'month', hcap.birthstartmonth,
          'day', hcap.birthstartday
        ),
      ),
  )

```

```

        'dateDeath', json_build_object(
            'year', hcap.deathstartyear,
            'month', hcap.deathstartmonth,
            'day', hcap.deathstartday
        )
    ))
FROM documentcoauthorperson dcap
JOIN historicalperson hcap
    ON hcap.id = dcap.authorperson_id
    AND dcap.validuntil > NOW()
    AND hcap.validuntil > NOW()
WHERE dcap.document_id = d.id
) AS coauthors,
(
    SELECT
        jsonb_agg(jsonb_build_object(
            'personId', hap.personid,
            'surname', hap.surname,
            'firstname', hap.firstname,
            'dateBirth', json_build_object(
                'year', hap.birthstartyear,
                'month', hap.birthstartmonth,
                'day', hap.birthstartday
            ),
            'dateDeath', json_build_object(
                'year', hap.deathstartyear,
                'month', hap.deathstartmonth,
                'day', hap.deathstartday
            )
        ))
    FROM documentaddresseepeople dap
    JOIN historicalperson hap
        ON hap.id = dap.addresseepeople_id
        AND dap.validuntil > NOW()
        AND hap.validuntil > NOW()
    WHERE dap.document_id = d.id
) AS addressees,
sc.city, d.documentcategory, d.ispublishedindb, d.createdat,
d.modifiedat, d.validuntil
FROM document d
LEFT JOIN sitecity sc ON sc.id = d.city_id;

```

Zusätzlich werden noch einige Indexe hinzugefügt, für eine bessere Performance bei der Abfrage, wie in 5.5 zu sehen.

Listing 5.5: SQL Materialized View

```

CREATE INDEX idx_searchdocument_documentid
    ON searchdocument (documentid);

CREATE INDEX idx_searchdocument_author_surname_firstname
    ON searchdocument ((author->>'surname'), (author->>'firstname'));

```

```

CREATE INDEX idx_searchdocument_startdate
  ON searchdocument (startyear, startmonth, startday);

CREATE INDEX idx_searchdocument_addressees_first_entry
  ON searchdocument
    ( ((addressees->0->>'surname')::text)
      , ((addressees->0->>'firstname')::text));

CREATE INDEX idx_searchdocument_city
  ON searchdocument (city);

CREATE INDEX idx_searchdocument_documentcategory
  ON searchdocument (documentcategory);

```

| # | Aufrufzeit (ms) | | | Queries (ms) | | Memory (MB) | | | Render (ms) | | | DB-load (ms) | | |
|---|-----------------|-----|-----|--------------|-----|-------------|-------|------|-------------|-----|-----|--------------|-----|-----|
| | min | avg | max | #-avg | avg | start | stop | diff | min | avg | max | min | avg | max |
| 1 | 203 | 315 | 808 | 17.8 | 3.0 | 851.4 | 883.9 | 32.5 | | | | | | |
| 2 | 154 | 172 | 187 | 9.0 | 2.2 | 883.2 | 887.0 | 3.8 | | | | | | |
| 3 | 145 | 151 | 163 | 9.0 | 2.8 | 887.7 | 895.3 | 7.6 | | | | | | |
| 4 | 132 | 143 | 152 | 9.0 | 2.8 | 896.0 | 900.0 | 4.0 | | | | | | |
| 5 | 121 | 125 | 132 | 9.0 | 2.4 | 900.6 | 901.0 | 0.4 | | | | | | |

Tabelle 5.9: Messung mit Materialized View

Wie in Tabelle 5.9 zu sehen, bringt die Verwendung der Materialized View eine Verbesserung in verschiedenen Punkten. Zum einen ist eine Verbesserung der Aufrufzeiten zu erkennen, zusätzlich fällt der Speicheranstieg weniger stark aus. Die Verbesserung der Aufrufzeiten lässt sich zusätzlich erklären, dass hier nun nur noch vier statt der 6 Abfragen an die Datenbank gestellt werden, da einzelabfragen für die Adressen der Personen und der Koautoren komplett entfallen.

Nach dem der Quellcode nochmal untersucht wurde, konnte man feststellen, dass bei jeder Anfrage die gleiche Bedingung benötigt wurde. Da die Sicht nun explizit für diese Anfrage geschaffen wurde, wurde die Bedingungen nun direkt in Sicht mit integriert. Dies bedeutet eine Erweiterung der Sicht aus 5.4 um 5.6 und das Entfernen der Parameter aus dem SQL-Anfragen im Java-Code.

Listing 5.6: SQL Materialized View Erweiterung

```

WHERE d.validuntil > NOW()
AND   d.ispublishedindb = true;

```

Nach dem Anpassungen haben sich dann die Werte aus 5.10 ergeben.

Da bei der Materialized View das Laden der Daten und das Wandeln in die Java-Objekte getrennt programmiert wurde, können hier eigene Zeitmessungen für die zwei Schritte eingebaut werden. Hierfür wird die Zeit vor dem *map*-Aufruf und der *map*-Aufruf gemessen. Für den ersten Aufruf, wurde ein *SearchDocument* Objekt erzeugt und immer dieses Objekt zurückgegeben. Damit wurde erst mal überprüft, wie lange das Ermitteln der Daten und das

| # | Aufrufzeit (ms) | | | Queries (ms) | | Memory (MB) | | | Render (ms) | | | DB-load (ms) | | |
|---|-----------------|-----|-----|--------------|-----|-------------|-------|------|-------------|-----|-----|--------------|-----|-----|
| | min | avg | max | #-avg | avg | start | stop | diff | min | avg | max | min | avg | max |
| 1 | 241 | 348 | 859 | 16.8 | xxx | 896.0 | 932.4 | xxxx | 232 | 331 | 803 | 132 | 174 | 334 |
| 2 | 164 | 194 | 225 | 9.0 | xxx | 933.3 | 935.9 | xxxx | 154 | 185 | 215 | 79 | 99 | 117 |
| 3 | 147 | 161 | 179 | 9.0 | xxx | 935.8 | 938.8 | 3.0 | 139 | 152 | 167 | 68 | 77 | 86 |
| 4 | 135 | 145 | 183 | 9.0 | xxx | 939.4 | 936.0 | -3.4 | 127 | 137 | 174 | 70 | 73 | 75 |
| 5 | 126 | 137 | 154 | 9.0 | xxx | 936.1 | 939.1 | 3.0 | 118 | 129 | 143 | 66 | 72 | 79 |

Tabelle 5.10: Messung mit erweiterter Materialized View

durcharbeiten der Ergebnisse bestimmt. Hierbei lagen die Zeiten bei circa 1 ms für das reine Datenladen und 3 ms für den Aufruf der *map*-Funktion. Sobald mal innerhalb der *map*-Funktion pro Eintrag ein Objekt erzeugt, noch ohne eine Konvertierung der ermittelten Daten in das Objekt, steigt die Laufzeit schon auf 54 ms. Wenn man nun noch die Konvertierung der Daten wieder einbaut, steigt die Laufzeit nochmal auf nun 82 ms. Dies zeigt, alleine das Erzeugen der Objekt kostet die meiste Zeit.

Bei der Verwendung des Hints *openjpa.FetchPlan.FetchBatchSize* kann die Abfrage enorm verschlechtern. Wenn dieser Wert zu klein oder groß definiert ist, wird die Laufzeit verschlechtert. Bei einem zu großen Wert wird die Laufzeit der Datenbankabfrage auf circa 20 ms verlängert. Wenn der Wert zu gering gewählt ist, dann wird zwar die Laufzeit der Datenbankabfrage minimal verkürzt, aber die *map*-Funktion wird dadurch verlängert.

Das Aktivieren der Cache-Optionen wie in 5.2 oder in ?? dargestellt, haben keine Auswirkung auf die Performance. Dies ist dadurch erklärbar, da keine Objekte durch das OpenJPA-Framework erstellt werden, sondern erst in der *map*-Funktion des eigenen Codes.

Nun wird noch geprüft, welche Performance das Parsen der Json-Informationen im Server benötigt. Im ersten Schritt wird die Parse-Funktion auskommentiert und die Seite nochmal aufgerufen. Durch diese Umstellung fällt die Laufzeit der Datenermittlung auf circa 4 ms ab. Nun muss noch geprüft werden, welche Zeit nun der Client zum Parsen der **Json!** (**Json!**)-Daten benötigt. Hierfür werden die Daten in einem versteckten **span**-Element hinterlegt, wie es im Beispiel 5.7 zu sehen ist. Die hinterlegte Cascading Style Sheets (CSS)-Klasse ist zum Auffinden der Elemente für den späteren Javascript. Das **ajax**-Element im Beispiel ist notwendig, damit bei einem Seitenwechsel die neu übertragenen Elemente in eine lesbare Form gebracht werden.

Listing 5.7: DataTable mit Json

```

<p:ajax event="page" oncomplete="convertJsonData()"/>
<p:column id="author" headerText="#{lang.List_Docs_Author}" sortable="
  true" sortBy="#{myObj.surname}">
  <h:outputText styleClass="json-convert" style="display: none;" value
    ="#{myObj.authorJson}"/>
</p:column>
<p:column id="Addressee"
  headerText="#{lang.List_Docs_Addressee}"
  sortable="true"

```



```

        sortBy="#{(myObj.addresseePersonSet!=null and myObj.
        addresseePersonSet.size() > 0)?myObj.addresseePersonSet
        [0].addresseePerson:''}">
    <h:outputText styleClass="json-convert" style="display: none;" value
    ="#{myObj.addresseeJson}" data="abc"/>
</p:column>

```

Um nun die übertragenen **Json!**-Daten in eine darstellbare Form zu bringen, benötigt man noch eine JavaScript-Funktion. Diese Funktion 5.8 wird ermittelt erst alle versteckten Elemente, parsed den Inhalt und erstellt neue HyperText Markup Language (HTML)-Elemente mit dem darzustellenden Inhalt. Zusätzlich wird noch eine Zeitmessung mit eingebaut, um die Laufzeit am Client für das Rendern in der Konsole anzuzeigen. Die Funktion wird nun direkt nach dem die Webseite fertig geladen wurde gerufen.

Listing 5.8: Wandeln von Json nach Html

```

function isEmpty(str) {
    return (str === null) || (str === undefined) || (typeof str === "
        string" && str.length === 0);
}
function convertJsonData() {
    let $jsonObj = $(".json-convert")
        , start = new Date()
        ;

    $.each($jsonObj, function() {
        let json = this.innerHTML
            , strEmpty = (json === null) || (typeof json === "string" && json.
                length === 0)
            , jsonDat = strEmpty ? null : JSON.parse(json)
            ;
        if(!strEmpty) {
            let res = ""
                , $that = $(this)
                , $par = $that.parent()
                ;
            $.each(jsonDat, function() {
                let hasOnlyOne = isEmpty(this.surname) || isEmpty(this.firstname
                    )
                    , pseudonymExists = !isEmpty(this.pseudonym)
                    , namePart = "<span>" + (hasOnlyOne ? this.surname + this.
                        firstname : this.surname + ", " + this.firstname) + "</
                        span><br/>"
                    , pseudoPart = pseudonymExists ? "<span class='w3-small w3-
                        text-dark-gray'>" + this.pseudonym + "</span><br/>" : ""
                    ;
                res += namePart + pseudoPart;
            });
            $par.append(res);
        }
    });
}

```

```

let end = new Date()
    , diff = (end - start)
    ;
console.log(Math.round(diff) + " ms");
}

$(document).ready(function() {
    convertJsonData();
});

```

Da nun am Client Code ausgeführt wird, nachdem die Daten übertragen wurden, kann nicht mehr alles über das Script durchgeführt werden. Daher werden nun die Laufzeiten am Server und am Client zusammenaddiert. Im Schnitt benötigt der Aufruf auf der Serverseite nun 70 ms und am Client sind es circa 13 ms. Dies bedeutet addiert kommt man mit dieser Lösung auf eine kürzere Laufzeit und weniger Last am Server.

TODO: Hier könnte man auch den Query-Cache nochmal verwenden, da die anfragen nun fix wären

TODO: Grundlagen zur Materialized-View noch hinterlegen

5.9 STATISCHE WEBSEITEN

Wenn man die Dokumentenliste als statische Webseiten ablegt, werden die Zugriffszeiten sehr kurz sein. Darüber hinaus funktionieren in statische Webseiten aber keine Suche oder eine Sortierung. Die Sortierung könnte durch das erstellen von statischen Seite aller Möglichkeiten der Sortierung emuliert werden, diese würde den notwendigen Speicherbedarf der Webseite vervielfachen. Für die Suchanfragen ist dies nicht mehr möglich, da nicht alle Suchanfragen vorher definiert werden können.

Die Umstellung der Suche auf Client!=Basis wäre noch eine Möglichkeit, dafür benötigen die Clients entsprechend Leistung und es muss eine Referenzdatei erstellt werden, die alle Informationen über die Dokumente beinhaltet, nach der gesucht werden kann.

Daher ist eine Umstellung auf statische Webseiten nicht sinnvoll.

5.10 CLIENT BASIERTE WEBSEITEN

Als weitere Möglichkeit könnte man die Webseite so umbauen, dass die Daten erst im Nachgang über eine AJAX-Anfrage ermittelt und die Sortierung und Aufteilung im Client durchgeführt wird. Hierbei wird aber je nach Datenmenge ein großer Speicher am Client benötigt und die Rechenleistung wird auf den Client verschoben.

Dies wiederum ist ein Vorteil für den Serverbetreiber, da durch die Verschiebung weniger Rechenleistung am Server benötigt wird. Gleichzeitig würde man damit wiederum schwächere Clients, wie Smartphones, ausperren, da bei diesem die notwendige Rechenleistung fehlt, um die Webseite in annehmbarer Zeit darzustellen.

EVALUIERUNG

Nun werden die durchgeführten Anpassungen anhand ihre Effektivität betrachtet und unter welchen äußeren Einflüssen diese eine Optimierung darstellen. Weiterhin werden die Nachteile der Anpassungen überprüft und und bei der Betrachtung der Effektivität mit beachtet.

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

6.1 NUTZERUMFRAGE

Zusätzlich war noch eine Befragung unter den Benutzer und den Entwicklern geplant. Da hierfür nur fünf Personen zur Verfügung stehen, ist dies nicht zielführend. Daher ist die sinnvolle Alternative ein rein technischer Ansatz, der gewählt wurde.

6.2 STATISCHE WEBSEITEN

TODO: prüfen wohin damit, hier oder odch in 5 lassen

Wenn man die Dokumentenliste als statische Webseiten ablegt, werden die Zugriffszeiten sehr kurz sein. Darüber hinaus funktionieren in statische Webseiten aber keine Suche oder eine Sortierung. Die Sortierung könnte durch das erstellen von statischen Seite aller Möglichkeiten der Sortierung emuliert werden, diese würde den notwendigen Speicherbedarf der Webseite vervielfachen. Für die Suchanfragen ist dies nicht mehr möglich, da nicht alle Suchanfragen vorher definiert werden können.

Die Umstellung der Suche auf Client!=Basis wäre noch eine Möglichkeit, dafür benötigen die Clients entsprechend Leistung und es muss eine Referenzdatei erstellt werden, die alle Informationen über die Dokumente beinhaltet, nach der gesucht werden kann.

Daher ist eine Umstellung auf statische Webseiten nicht sinnvoll.

6.3 CLIENT BASIERTE WEBSEITEN

TODO: prüfen wohin damit, hier oder odch in 5 lassen

Als weitere Möglichkeit könnte man die Webseite so umbauen, dass die Daten erst im Nachgang über eine AJAX-Anfrage ermittelt und die Sortierung und Aufteilung im Client durchgeführt wird. Hierbei wird aber je nach Datenmenge ein großer Speicher am Client benötigt und die Rechenleistung wird auf den Client verschoben.

Dies wiederum ist ein Vorteil für den Serverbetreiber, da durch die Verschiebung weniger Rechenleistung am Server benötigt wird. Gleichzeitig würde man damit wiederum schwächere Clients, wie Smartphones, aussperren, da bei diesem die notwendige Rechenleistung fehlt, um die Webseite in annehmbarer Zeit darzustellen.

6.4 STATISTIKEN IM POSTGRESQL AUSWERTEN

6.5 VERGLEICH DER ERGEBNISSE VOR UND NACH DER OPTIMIERUNG

6.5.1 *Caching im OpenJPA*

Bei der Verwendung des OpenJPA-Caches gibt es einige Verbesserungen in der Geschwindigkeit zu sehen. Die Höhe der Optimierungen hängt stark von der gewählten Cache-Größe und der aufgerufenen Webseiten ab. Solange die Anfragen sich auf die gleichen Objekte beziehen und diese alle im Cache hinterlegt werden können, fällt die Optimierung entsprechend hoch aus. Sobald bei den Anfragen aber häufig die zu ermittelnden Objekte sich unterscheiden und alte Objekte wieder aus dem Cache entfernt werden, fällt die Performance-Verbesserung immer geringer aus.

Das Entfernen der Objekte kann zwar umgangen werden, indem die häufig abgefragten Objekte gepinnt werden, was aber den Speicherbedarf noch weiter erhöht, da diese Objekte nicht in die Zählung der Cache-Objekte beachtet werden. Was uns direkt zum größten Nachteil dieser Caches kommen lässt, die notwendige Speichermenge die ständig zur Verfügung gestellt werden muss. Damit ist immer ein gewisser Grundbedarf notwendig, da sich der Speicher bis zum eingestellten Grenzwert aufbaut und dann nicht mehr entleeren wird. Gerade bei kleiner dimensionierten Servern stellt dies ein größeres Problem dar, da dann weniger Speicher für die anderen laufenden Programme, wie dem Datenbankmanagementsystem, zur Verfügung steht.

Hierbei ist aber noch zu beachten, dass die Optimierung durch den Cache nicht die Laufzeit der Abfragen in der Datenbank enorm verringert hat, sondern die Laufzeit beim Erstellen der Objekte im *OpenJPA*-Framework. Dies sieht man sehr gut schon bei der ersten Messung, wie in ???. Hierbei werden die Laufzeit in der Datenbank im Schnitt um circa 5 ms reduziert, aber die komplette Webseite wird fast 100 ms schneller an den Client ausgeliefert. Dies ist nur dadurch erklärbar, dass das Erstellen und mit den Datenwerten zu befüllen mehr Zeit kostet, als das Objekt aus dem Cache zu ermitteln und zurückzugeben.

Daher ist die Verwendung des OpenJPA-Cache nur in Verbindung mit einem größer dimensionierten Server gut verwendbar, wenn der Großteil der Objekte im Cache gehalten werden kann. Bei Bedarf sollten die häufig frequentierten Objekte explizit im Cache aufgenommen und angepinnt werden.

6.5.2 *Cached Queries*

Die Optimierung über die gespeicherten Anfragen brachte keine Verbesserung hervor. Dies ist dadurch erklärbar, dass für diese Art nur Anfragen verwendet werden, die keinerlei Bedingungen besitzen. Da in diesem Fall in der Tabelle auch noch nicht freigegebene und ungültige Datensätze gespeichert sind, müssen diese vor dem Übertragen herausgefiltert werden. Dies ist der Grund warum diese Anfragen in diesem Cache nicht gespeichert werden.

Dadurch ist dieser Cache für eine Performance-Verbesserung in unseren Fall nicht verwendbar.

6.5.3 *Caching in JPA*

6.5.4 *Caching in EJB*

6.5.5 *Abfragen mit JPQL und Criteria API*

Bei dem Vergleich zwischen den 2 Abfragemöglichkeiten der **JPQ!** (**JPQ!**) und der Criteria API konnte in der Art der Abfragen kein Unterschied dargestellt werden. Die Abfragen der beiden Systeme sind auf der Datenbankseite komplett identisch. Auch in der Übertragung der Daten aus der Datenbank in die Java-Objekte konnte keine Unterschied in der Art und Geschwindigkeit festgestellt werden.

Ebenfalls sind die Möglichkeiten über der Optimierung über Hints identisch. In beiden Fällen, haben die meisten Hints keine nennenswerten Einfluss auf die Laufzeit der Abfragen und Übertragung in die Java-Objekte. Das sinnvolle setzen von `OptimizeResultCount`, der `FetchSize` sowie der `FetchBatchSize` hilft dem Framework die Bearbeitung der Anfrage effizient abzuarbeiten, konnte aber in den gemessenen Laufzeiten nicht verifiziert werden.

Anders ist dies mit dem Einstellungen für `EagerFetchMode`, welche definiert wie die Daten für abhängige Klasse ermittelt werden. Bei Umstellung auf `parallel` konnte für die Ermittlung der Dokumente einiges an Performance gewonnen werden. Das liegt daran, dass nun für die abhängigen Objekte, wie den Koautoren, nicht pro Dokument eine Anfrage an die Datenbank gestellt wird, sondern es werden alle Koautoren für die ermittelten Dokumente auf einmal ermittelt. Die Zuordnung der Koautoren zu dem Dokument wird dann nun im Framework und nicht mehr durch die Datenbank durchgeführt. Diese Abarbeitung spart viele einzelne Abfragen und somit auch den entsprechend Overhead im Framework.

Aufgrund dessen ist die Entscheidung der Technik für die Performance irrelevant und es kann das genutzt werden, was für jeweiligen Einsatzzweck besser beziehungsweise einfacher zu programmieren ist. Das setzen der richtigen Hints wiederum ist in beiden Fällen äußerst wichtig. Explizit der `EagerFetchMode` muss vorher darüber nachgedacht werden, wie viele abhängi-

ge Objekttypen es zu dieser Klasse gibt, welche dazu geladen werden sollen und von welcher Anzahl an Objekte ausgegangen werden kann. Gerade bei ein größeren Anzahl lohnt es sich den Hint auf *parallel* zu setzen. Gleiches gilt den Hint *SubclassFetchMode*, dieser steuert die Abfragen im falle von abgeleitet Klassen.

6.5.6 *Materialized View*

TODO: hier weiter machen

ZUSAMMENFASSUNG UND AUSBLICK

TODO: hier eine kurze Zusammenfassung über 2-3 Seiten

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() {
    {
        local maxLen=0
        for url in ${@:2}; do
            local size=${#url}
            if [[ $size -gt $maxLen ]]
            then maxLen=$size
            fi
        done

        printf "%-${maxLen}s  %4s %5s %9s %9s %9s %9s\n" "URL" "
            Runs" "StDev" "Min (ms)" "Avg (ms)" "Max (ms)" "1st (ms)"
        for url in ${@:2}; do
            get_statistics $url $1 $maxLen
        done
    } #| column -s $'\t' -t
}

get_statistics() {
    # Initialize the variables
    local first=-1
    local min=1000000000
    local max=0
    local dur=0
    local durQ=0
    local spin=0
    local activeSpinner=0
    local spiner=('-' '\ ' '| ' '/')

    printf "%-${maxLen}s  " $1
    if [[ $activeSpinner -ne 0 ]]
    then printf " "
    fi
    #echo -ne "$1\t "
}
```

```

# repeat for the defined counts the url calling
for i in $(seq 1 $2); do
    if [[ $activeSpinner -ne 0 ]]
        then echo -ne "\b${spiner[$spin]}"
    fi
    spin=$(( (spin+1) % 4 ))
    local gp=$((get_posts $1)/1000000) # from ns to ms
    if [[ $first -lt 0 ]]
        then first=$gp
    fi
    if [[ $gp -gt $max ]]
        then max=$gp
    fi
    if [[ $gp -lt $min ]]
        then min=$gp
    fi
    dur=$(( $dur + $gp ))
    durQ=$(( $durQ + (($gp * $gp)) ))
done

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

# output the statistic values
if [[ $activeSpinner -ne 0 ]]
    then echo -ne "\b"
fi
printf "%+4s %+5s %+9s %+9s %+9s %+9s\n" $2 $stdev $min $avg $max
    $first
#echo -e "\b$2\t$stdev\t$min\t$avg\t$max\t$first"
}

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 %-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) }'
}

print_docker_process() {

```

```
    sudo docker stats --no-stream
}

# 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/document/listsearch.xhtml"
)

#print_process
print_docker_process
echo ""

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

echo ""
##print_process
print_docker_process
```

DOCKER KONFIGURATION

Da die Leistung des Computers zu hoch ist, um vergleichbare Zahlen bei den Datenbank-Anfragen ermitteln zu können, wurde der Postgres-Server sowie der Payara-Server in einen Docker-Container verpackt und die Leistung limitiert.

Listing B.1: Docker-Compose

```
services:
  db:
    image: postgres
    container_name: postgreddb
    restart: always
    hostname: postgres
    ports:
      - "5432:5432"
    environment:
      POSTGRES_USER: <username>
      POSTGRES_PASSWORD: <password>
      POSTGRES_DB: <dbname>
    volumes:
      - ./data:/var/lib/postgresql/data
    deploy:
      resources:
        limits:
          cpus: '0.30'
          memory: 500m
  ws:
    image: payara/server-full
    container_name: payara
    ports:
      - "4848:4848"
      - "8080:8080"
      - "8181:8181"
      - "9009:9009"
    volumes:
      - ./payara/postgresql-42.7.3.jar:/opt/payara/appserver/glassfish/domains/domain1/lib/postgresql-42.7.3.jar
      - ./payara/logs:/opt/payara/appserver/glassfish/domains/domain1/logs/
```

```
- ./payara/config:/opt/payara/appserver/glassfish
  /domains/domain1/config/
- ./payara/applications:/opt/payara/appserver/
  glassfish/domains/domain1/applications/
deploy:
  resources:
    limits:
      cpus: '2.00'
      memory: 2g
links:
  - db
```

AUFRUF SKRIPT

Um die Messungen etwas zu vereinfachen wurde ein Skript erstellt um die Aufrufe gesammelt durchzuführen. Um die Messungen durchzuführen werden die Befehl, wie in C.2 dargestellt aufgerufen. Durch die nummerierten Präfixe können im Nachgang über die *pgBadger*-Berichte die SQL-Abfragen verglichen werden. Wichtig hierbei ist noch, dass vor dem *measrun*-Aufruf überprüft wird, ob die Docker-Container gestartet und initialisiert sind. Wenn dies nicht der Fall ist, laufen die Abfragen ins leere. Am einfachsten ist das, wie dargestellt, über die Statistik von Docker zu ermitteln. Nun wird überwacht, das die CPU-Auslastung auf ein niedriges Level fällt, danach kann das Skript für die Messung gerufen werden.

Listing C.1: Calling Script

```
#!/bin/bash

payara_path="/opt/docker/payara"
postgres_path="/var/lib/postgres"
postgres_path="/opt/docker"
postgres_data_path="$postgres_path/data"
postgres_log_path=$postgres_data_path/log

payara_config="$payara_path/config/domain.xml"
domain_log="$payara_path/logs/server.log"
script_path="/opt/docker/timing.sh"
pgbadger_out="/opt/docker/pgreport"
report_postfix=""
report_postno=""
report_postmax="5"
docker_name=dcpgbatch

COMPOSE_FILE=/opt/docker/docker-compose.yaml

gflog() {
    echo "follow the log: $domain_log"
    tail -f $domain_log
}
gflogrm() {
    echo "remove the log: $domain_log"
    rm "$domain_log"
}
gfconf() {
    nvim "$payara_config"
}
gfscript() {
    gflogcount=$(cat "$domain_log" | wc -l)
```

```

outPath=$pgbadger_out$report_postfix/bash.out
touch "$outPath"
echo "" >>"$outPath"
echo "===== " >>
"$outPath"
echo "calling number $report_postno / $report_postmax" >>"$outPath"
echo "" >>"$outPath"
bash $script_path | tee -a "$outPath"
gflastlog=$(cat "$domain_log" | wc -l)
gfcnt=$((gflastlog - $gflgcount))
printf "\n%-20s %5s %9s %9s %9s %9s\n" "function" "Runs" "Min (ms)"
" " "Avg (ms)" "Max (ms)" "1st (ms)" >>"$outPath"

tail -$gfcnt $domain_log | awk '/PerformanceCounter-create \view/
{ print $3 }' | awk 'BEGIN {min=500,max=0} {sum += $1; cnt +=
1}; first == "" { first = $1 }; min > $1 { min = $1 } max < $1 {
max = $1 } END { if (cnt < 1) {cnt = -1}; printf("%-20s %5i %9.3f
%9.3f %9.3f\n", "view-create-list", cnt, min, sum/cnt, max,
first) }' >>"$outPath"
tail -$gfcnt $domain_log | awk '/PerformanceCounter-build \view/
{ print $3 }' | awk 'BEGIN {min=500,max=0} {sum += $1; cnt +=
1}; first == "" { first = $1 }; min > $1 { min = $1 } max < $1
{ max = $1 } END { if (cnt < 1) {cnt = -1}; printf("%-20s %5i %9.3
f %9.3f %9.3f %9.3f\n", "view-buildr-list", cnt, min, sum/cnt, max
, first) }' >>"$outPath"
tail -$gfcnt $domain_log | awk '/PerformanceCounter-loadData \view/
{ print $3 }' | awk 'BEGIN {min=500,max=0} {sum += $1; cnt +=
1}; first == "" { first = $1 }; min > $1 { min = $1 } max < $1 {
max = $1 } END { if (cnt < 1) {cnt = -1}; printf("%-20s %5i %9.3f
%9.3f %9.3f %9.3f\n", "view-loaddt-list", cnt, min, sum/cnt, max,
first) }' >>"$outPath"
tail -$gfcnt $domain_log | awk '/PerformanceCounter-loadDataMap \view/
/ { print $3 }' | awk 'BEGIN {min=500,max=0} {sum += $1; cnt +=
1}; first == "" { first = $1 }; min > $1 { min = $1 } max < $1 {
max = $1 } END { if (cnt < 1) {cnt = -1}; printf("%-20s %5i %9.3f
%9.3f %9.3f %9.3f\n", "view-loaddm-list", cnt, min, sum/cnt, max,
first) }' >>"$outPath"
tail -$gfcnt $domain_log | awk '/PerformanceCounter-renderData \view/
{ print $3 }' | awk 'BEGIN {min=500,max=0} {sum += $1; cnt +=
1}; first == "" { first = $1 }; min > $1 { min = $1 } max < $1 {
max = $1 } END { if (cnt < 1) {cnt = -1}; printf("%-20s %5i %9.3f
%9.3f %9.3f %9.3f\n", "view-rdrdat-list", cnt, min, sum/cnt, max,
first) }' >>"$outPath"
tail -$gfcnt $domain_log | awk '/PerformanceCounter-renderLoad \view/
{ print $3 }' | awk 'BEGIN {min=500,max=0} {sum += $1; cnt +=
1}; first == "" { first = $1 }; min > $1 { min = $1 } max < $1 {
max = $1 } END { if (cnt < 1) {cnt = -1}; printf("%-20s %5i %9.3f
%9.3f %9.3f %9.3f\n", "view-rdrlst-list", cnt, min, sum/cnt, max,
first) }' >>"$outPath"
tail -$gfcnt $domain_log | awk '/PerformanceCounter-render \view/
{ print $3 }' | awk 'BEGIN {min=500,max=0} {sum += $1; cnt +=
1}; first == "" { first = $1 }; min > $1 { min = $1 } max < $1 {

```

```

        max = $1 } END { if (cnt < 1) {cnt = -1}; printf("%-20s %5i %9.3f
        %9.3f %9.3f %9.3f\n", "view-render-list", cnt, min, sum/cnt, max,
        first) }' >>"$outPath"
    }

pginit() {
    sudo chmod g+x $postgres_path
    sudo chmod g+x $postgres_data_path
    sudo chmod g+rx $postgres_log_path
}

pglogls() {
    echo "show postgresql logfiles"
    ls $postgres_log_path/ -lhtc
}

pglogrm() {
    cnt=${1:-10}
    cntTail=$((cnt + 1))
    echo "remove old postgresql logfiles from $(ls $postgres_log_path/ -tc
    | wc -l) until $cnt (~${cnt}oo MB) $cntTail"
    ls $postgres_log_path/ -tc | tail -n +$cntTail | xargs -r -I{} sudo rm
    "$postgres_log_path/{}"
}

pglog() {
    pg_log=$(ls $postgres_log_path/ -tc --ignore '*log' | head -n 1)
    echo "follow the log: $postgres_log_path/$pg_log"
    tail -n 3 -f $postgres_log_path/$pg_log
}

pgconf() {
    sudo nvim "${postgres_path}/postgresql.conf"
}

pgrp() {
    mkdir -p $pgbadger_out$report_postfix
    mkdir -p $pgbadger_out$report_postfix$report_postno
    outPath=$pgbadger_out$report_postfix/bash.out
    touch "$outPath"
    echo "" >>"$outPath"
    pgbadger -X -I -f jsonlog -j 10 -0
    $pgbadger_out$report_postfix$report_postno $postgres_log_path/
    postgresql-*.json 2>&1 | tee -a "$outPath"
}

pgrpres() {
    if [[ "$report_postfix" == "" ]]; then
        rm -R $pgbadger_out
    else
        rm -R $pgbadger_out$report_postfix*
    fi
}

pgrpout() {
    echo "Show bash output"
    outPath=$pgbadger_out$report_postfix/bash.out
    cat "$outPath"
}

```



```

dcreate() {
    sudo docker compose -f $COMPOSE_FILE create --force-recreate
}
dcconf() {
    nvim $COMPOSE_FILE
}
dcstart() {
    sudo docker compose -f $COMPOSE_FILE start
    sleep 2
    pginit
}
dcstop() {
    sudo docker compose -f $COMPOSE_FILE stop
}
dcres() {
    sudo docker compose -f $COMPOSE_FILE down
}
dcstats() {
    sudo docker stats
}
meascall() {
    gfscrip
    pgrp
    pglogrm
}
measrun() {
    for i in $(seq 1 $report_postmax); do
        report_postno=$i
        meascall
    done
}
for name in "$@"; do
    case $name in
        -rppf=*) report_postfix="{name#*=}" ;;
        -rppn=*) report_postno="{name#*=}" ;;
        -rppm=*) report_postmax="{name#*=}" ;;
    )
    gflog) gflog ;;
    gflogrm) gflogrm ;;
    gfconf) gfconf ;;
    gfscrip) gfscrip ;;
    gfrestart) pgrpinit ;;
    pginit) pginit ;;
    pglogls) pglogls ;;
    pglogrm) pglogrm ;;
    pglog) pglog ;;
    pgconf) pgconf ;;
    pgrestart) pgrestart ;;
    pgrp) pgrp ;;
    pgrpres) pgrpres ;;
    pgrpout) pgrpout ;;
    dcinit) dcreate ;;
    dcconf) dconf ;;

```

```

dcstart) dcstart ;;
dcstop) dcstop ;;
dcres) dcres ;;
dcstats) dcstats ;;
measinit)
    pginit
    pgrpres
    pglogrm 0
    gflogrm
    dcreate
    dcstart
    ;;
measres)
    dcstop
    pgrpres
    pglogrm 0
    gflogrm
    dcstart
    pgrp
    pglogrm
    ;;
meascall) meascall ;;
measrun) measrun ;;
help)
    echo "CALLING: $0 <function> [ <function>]"
    echo "The overview of the functions of this script."
    echo "It is allowed to enter multiple functions in one execute,"
    echo "that would be called serialized."
    echo "ATTENTION: parameter must be defined in front of the commands!"
    "
    echo ""
    echo "*** parameter ***"
    echo "  -rppf=<val>  Postfix name for the report-folder (used by
                    gfscrip, pgrp, pgrpres, measres, meascall)"
    echo "  -rppn=<val>  Postfix number for the report-folder (used by
                    pgrp, measres, meascall)"
    echo "  -rppm=<val>  Count of calling meascall (used by measrun)"
    echo ""
    echo "*** glassfish ***"
    echo "  gflog      Show and follow the log of the glassfish server
                    with \$domain_name"
    echo "  gflogrm   Remove the log file of the glassfish server with
                    \$domain_name"
    echo "  gfconf    Open the configuration file from glassfish server"
    echo "  gfscrip   Calls the testscript for the website"
    echo ""
    echo "*** postgresql ***"
    echo "  pginit    Initialize the folder for postgresql log-folder"
    echo "  pglogls   Show the current content of the log-folder from
                    postgresql"
    echo "  pglogrm   Clean the log-folder from postgresql, the newest
                    20 files are sill available"

```

```

echo "  pglog      Show and follow the last log from postgresql"
echo "  pgconf     Open the configuration file from postgresql"
echo "  pgrestart  Restart the postgresql"
echo "  pgrp       Generate the pgbadger report from postgresql log
            files"
echo "  pgrpout    Show the saved output of the bash scripts"
echo "  pgrpres    Resetet the output of pgbadger"
echo ""
echo "*** docker ***"
echo "  dcinit     Create the both docker container"
echo "  dcconf     Open the configuration file for the docker
            container"
echo "  dcstart    Start the both docker container"
echo "  dcstop     Stop the both docker container stoppen"
echo "  dcres      Stop and remote the both docker container"
echo "  dcstats    Show the docker live statistik"
echo ""
echo "*** combine cmds ***"
echo "  measinit   Initialize everthing after start"
echo "  measres    reset data for new measuring"
echo "  meascall   execute one measure"
echo "  measrun    execute <ppm> calls of measure (default: 5 runs)"
;;
*)
echo >&2 "Invalid option $name"
exit 1
;;
esac
done

```

Listing C.2: Aufrufe des Unterstützungsscriptes

```

callscript.sh measinit
callscript.sh -rppf=_testname measres
callscript.sh dcstats
callscript.sh -rppf=_testname measrun

```

JSF PERFORMANCE STATISTICS SERVLET

Um die Cache-Informationen über einen Application Programming Interface (API)-Aufruf bereitzustellen, wird ein Servlet D.1 und ein Provider D.2 benötigt. Für die Aufnahme in der Routing, wird noch ein zusätzlicher Eintrag in der *web.xml* benötigt D.3.

Listing D.1: Performance Statistics Servlet

```
public class PerfStatServlet extends HttpServlet {
    private final static Logger m_Logger = Logger.getLogger(
        PerfStatServlet.class.toString());

    @Inject
    private PerfStatistics perfStatistics;

    public PerfStatServlet() {
        m_Logger.info("PerfStatServlet() Constructor");
    }

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse
        resp) throws ServletException, IOException {
        String openJPAStatistics = perfStatistics.getOpenJPAStatistics()
            ;
        String entityClasses = perfStatistics.getEntityClasses();

        resp.setStatus(HttpServletResponse.SC_OK);
        resp.setContentType(MediaType.APPLICATION_JSON);
        PrintWriter out = resp.getWriter();
        out.print("{ \"openJPAStatistics\": ");
        out.print(openJPAStatistics);
        out.print(", \"entityClasses\": ");
        out.print(entityClasses);
        out.print(" }");
    }
}
```

Listing D.2: Performance Statistics Provider

```
public interface PerfStatistics {
    String getOpenJPAStatistics();
    String getEntityClasses();
}

@Stateless
public class PerfStatisticsImpl implements PerfStatistics {
    @Inject
```

```

private EntityManagerJPAFactory entityManagerFactory;
private OpenJPAEntityManagerFactory openJPAEntityManagerFactory;

@PostConstruct
public void init() {
    openJPAEntityManagerFactory = OpenJPAPersistence.cast(
        entityManagerFactory.getEntityManager().
            getEntityManagerFactory());
}

@Override
public String getOpenJPAStatistics() {
    StoreCache cache = openJPAEntityManagerFactory.getStoreCache();

    CacheStatistics st = cache.getStatistics();
    return String.format("{ \ReadCount\: %d, \HitCount\: %d, \
WriteCount\: %d }", st.getReadCount(), st.getHitCount(), st
    .getWriteCount());
}

@Override
public String getEntityClasses() {
    StringBuilder res = new StringBuilder("{ \Entities\: [");

    EntityManagerFactory emf = openJPAEntityManagerFactory;
    EntityManager em = emf.createEntityManager();
    Cache cache = emf.getCache();
    for(EntityType<?> entityType : em.getMetamodel().getEntities())
    {
        Class<?> cls = entityType.getBindableJavaType();
        res.append(String.format("\%s\ ", cls.getCanonicalName())
            );
    }
    res.append("\\"], \Cache\: \");
    res.append(cache.toString());
    res.append("\ }");
    return res.toString();
}
}

```

Listing D.3: Einbindung Servlet

```

<servlet>
  <servlet-name>PerformanceStatisticServlet</servlet-name>
  <servlet-class>de.wedekind.servlets.PerfStatServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>PerformanceStatisticServlet</servlet-name>
  <url-pattern>/api/perfstat</url-pattern>
</servlet-mapping>

```

JSF PERFORMANCE MEASURE

Für die Protokollierung der Abläufe im JSF werden zwei Klassen benötigt. Die Factory E.2, wird die Wrapper-Klasse in die Bearbeitungsschicht eingeschleust. Diese Wrapper-Klasse E.1 beinhaltet dann die eigentliche Performance-Messung, inklusive der Ausgabe in die Log-Datei des *Glassfish*-Servers. Zusätzlich muss in der Konfiguration **faces-config.xml** noch angepasst werden, wie in E.3, um die Factory durch das System aufrufen zu lassen.

Listing E.1: Vdi Logger

```
public class VdlLogger extends ViewDeclarationLanguageWrapper {
    private static final Logger logger_ = Logger.getLogger(VdlLogger.
        class.getName());
    private final ViewDeclarationLanguage wrapped;

    public VdlLogger(ViewDeclarationLanguage wrapped) {
        this.wrapped = wrapped;
    }

    @Override
    public UIViewRoot createView(FacesContext context, String viewId) {
        long start = System.nanoTime();
        UIViewRoot view = super.createView(context, viewId);
        long end = System.nanoTime();
        logger_.severe(String.format("PerformanceCounter-create %s: %.6f
            ms", viewId, (end - start) / 1e6));
        return view;
    }

    @Override
    public void buildView(FacesContext context, UIViewRoot view) throws
        FacesException, IOException {
        long start = System.nanoTime();
        super.buildView(context, view);
        long end = System.nanoTime();
        logger_.severe(String.format("PerformanceCounter-build %s: %.6f
            ms", view.getViewId(), (end - start) / 1e6));
    }

    @Override
    public void renderView(FacesContext context, UIViewRoot view) throws
        FacesException, IOException {
        long start = System.nanoTime();
        super.renderView(context, view);
        long end = System.nanoTime();
        logger_.severe(String.format("PerformanceCounter-render %s: %.6f
            ms", view.getViewId(), (end - start) / 1e6));
    }
}
```

```

    }

    @Override
    public UIComponent createComponent(FacesContext context, String
        taglibURI, String tagName, Map<String, Object> attributes) {
        long start = System.nanoTime();
        UIComponent component = super.createComponent(context, taglibURI
            , tagName, attributes);
        long end = System.nanoTime();
        logger_.severe(String.format("PerformanceCounter-creatc $s: %.6f
            ms", taglibURI, (end - start) / 1e6));
        return component;
    }

    @Override
    public ViewDeclarationLanguage getWrapped() { return wrapped; }
}

```

Listing E.2: Vdi Logger Factory

```

public class VdlLoggerFactory extends ViewDeclarationLanguageFactory {
    private final ViewDeclarationLanguageFactory wrapped_;

    public VdlLoggerFactory(ViewDeclarationLanguageFactory
        viewDeclarationLanguage) {
        wrapped_ = viewDeclarationLanguage;
    }

    @Override
    public ViewDeclarationLanguage getViewDeclarationLanguage(String
        viewId) {
        return new VdlLogger(wrapped_.getViewDeclarationLanguage(viewId)
            );
    }

    @Override
    public ViewDeclarationLanguageFactory getWrapped() { return wrapped_
        ; }
}

```

Listing E.3: Einbindung Factory

```

<factory>
  <view-declaration-language-factory>
    de.wedekind.utils.VdlLoggerFactory
  </view-declaration-language-factory>
</factory>

```

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).
- [Con] *Config - Too small work_mem · pganalyze*. URL: https://pganalyze.com/docs/checks/settings/work_mem (besucht am 12.09.2024).
- [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>.