FernUniversität in Hagen

– Fakultät für Mathematik und Informatik –

Lehrgebiet Datenbanken und Informationssysteme

Analyse und Optimierung der Webseite des Wedekind Projektes

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, 01. Januar 2024

Marco Galster

ABSTRACT

TODO: Dies am Ende noch Ausfüllen!!!

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

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

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

`https://plg.uwaterloo.ca/~migod/research/beckOOPSLA.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/beckOOPSLA.html`

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

LISTINGS

ABKÜRZUNGSVERZEICHNIS

EJB Enterprise Java Beans

JSP Java Server Page

ORM Object Relational Mapping

SFSB Stateful Session-Bean

JPA Java Persistence API

1 EINLEITUNG

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

1.1 MOTIVATION

Die Frank-Wedekind-Bibliothek soll als Grundlage weiterer Forschungen dienen. Durch die weitere Forschung soll die literarhistorische und kulturgeschichtliche Wissen über die Kultur zwischen 1880 und 1918 erweitern. Um dieses Vorhaben umsetzen zu können muss die Anwendung dem Benutzer gerecht werden, damit diese verwendet wird und somit den Wissenstand entsprechend erweitert werden kann.

Um das Ziel der Akzeptanz zu erreichen und das sich die Bediener rein auf ihre Arbeit konzentrieren können, muss mit der Anwendung flüssig interagiert werden können. Entsprechend müssen die Wartzeiten auf ein minimum reduziert werden. Des Weiteren muss die Stabilität der Anwendung gesteigert werden.

1.2 AUSGANGSLAGE

TODO: hier die Grundlagen aus dem Expose? Braucht man die Motivation dann noch?

1.3 ZIEL DER ARBEIT

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

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

Hierbei ist auch ein Vergleich mit anderen Technologien angedacht.

1.4 GLIEDERUNG

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

Nachfolgenden wird im Kapitel 3 die Konzepte vorgestellt mit welchen man die Probleme ermitteln wird. Hierbei ist zusätzlich ein Blick auf andere Frameworks sinnvoll, um dort aus den bekannten Anomalien zu lernen und deren Lösungsansatz zu überprüfen ob diese angewandt werden kann.

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

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

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. Zusätzliche wird beschrieben wie ein weiteres Vorgehen durchgeführt werden kann.

2 GRUNDLAGEN

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

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

Die Datenanfragen werden über die *Enterprise Java Beans (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.

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 danach wieder gelöscht werden. Dies birgt allerdings den Nachteil, dass der *Persistenzkontext* sehr groß werden kann, wenn viele Entities in den *Persistenzkontext* geladen werden. Da dies häufig zu Speicher- und damit Performanz-Problemen [MW12, S. 79] führen kann, muss hier darauf geachtet werden, nicht mehr benötigte Entities aus dem *Persistenzkontext* zu lösen.

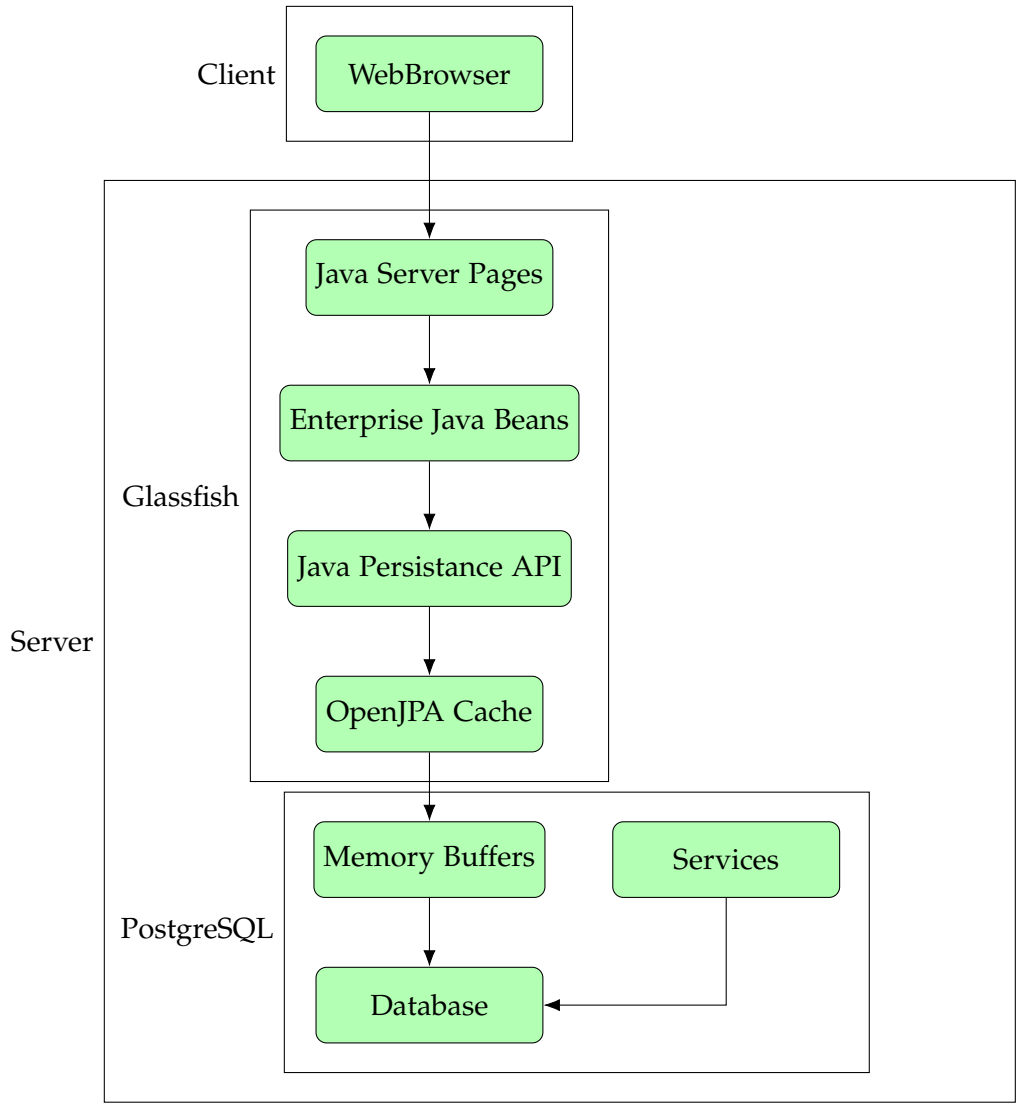

Abbildung 2.1: Ablauf einer Web-Anfrage

2.2 GLASSFISH - JAVA PERSISTANCE 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 Objekt erzeugt, aber noch nicht in den Cache überführt worden. Wenn diese in den Cache überführt worden sind, nehmen sie den Zustand *Verwaltet* ein. Ist das Objekt aus dem Cache und der Datenbank entfernt worden, nimmt es den Zustand *Gelöscht* an. *Losgelöst* ist der letzte Zustand, bei dem das Objekt aus dem Cache entfernt worden ist, aber nicht aus der Datenbank.

Eine Menge von Objekten wird als *Persistenzkontext* bezeichnet. Solange die Objekte dem *Persistenzkontext* zugeordnet sind, also den Zustand *Verwaltet* besitzen, werden diese auf Änderungen überwacht, um sie am Abschluss mit der Datenbank zu synchronisieren. In der Literatur wird hierzu der Begriff *Automatic Dirty Checking* verwendet [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 bzw. die Eigenschaften einer Abfrage und die zurückgelieferten Ids der Entities gespeichert. Bei einen erneuten Aufruf dieser Abfrage werden die referenzierten Objekte aus dem *Objekt-Cache* zurückgegeben. Bei veränderten referenzierten Entities wird der *Query-Cache* nicht genutzt und die betroffenen Abfragen werden unverzüglich aus dem *Query-Cache* entfernt [MW12, S. 316].

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

2.4 POSTGRESQL - MEMORY BUFFERS

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

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

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

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

2.5 POSTGRESQL - SERVICES

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

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

2.6 POSTGRESQL - ABFRAGEN

Für weitere Optimierungen werden anschließend die Anfragen einzeln überprüft. Hierfür ist es sinnvoll die Ausführungspläne der Abfrage zu analysieren [EH13, S. 252], die verschiedenen Plantypen und ihre Kosten zu kennen, sowie die angegeben 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 weißen 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.

3 KONZEPT

3.1 AUFBAU DER UMFRAGE

Die Umfrage wird über Email an die XX Personen verschickt. Als Basis für die Umfrage wird der aktuelle Prototyp unter verwendet. Hierbei wird die gleiche Umfrage für Bearbeiter und Benutzer versendet.

Die erste Frage ist zur Unterscheidung ob die Antworten von einen Bearbeiter oder von einem Benutzer kommt. Dies ist nur notwendig, um bei der Nachstellung zu unterscheiden welche Zugriffsrechte aktiv sind.

Die weiteren Fragen sind aufeinander aufgebaut. Hierbei wird zuerst überprüft, bei welchen Aktionen eine längere Wartezeit auftritt. Zusätzlich soll noch dazu angegeben werden, wie häufig dies auftritt, also ob dies regelmässig auftritt oder immer nur zu bestimmten Zeitpunkten. Des Weiteren wird die Information nachgefragt, ob die Probleme immer bei der gleichen Abfolge von Aktionen auftritt.

Die Anfrage wird im Anhang A dargestellt.

3.2 ALLGEMEINE BETRACHTUNG DES SYSTEMS

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

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

TODO: die anderen Speicher abarbeiten?

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

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

Da unter Linux der Arbeitsspeicher nicht mehr direkt freigegeben wird, ist hier die Page-Datei der wichtigere Indikator. Wenn dieses in Verwendung ist, dann benötigt die aktuell laufenden Programme mehr Arbeitsspeicher als vorhanden, 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.

TODO: Bespreibung der untersuchung von Glassfisch? oder lieber später

3.3 DAS VORGEHEN DER OPTIMIERUNG

TODO: Beschreibung der Untersuchung von Glassfisch? oder lieber später, oder doch eher umbennen in E

3.4 AKTUELLER AUFBAU DER SOFTWARE

3.5 VERGLEICH MIT ANDEREN TECHNOLOGIEN

Damit eine Umsetzung mit einer anderen Technologie umgesetzt werden kann, muss dies den kompletten Aufbau unterstützen, wie dies die Java Server Page (JSP) unterstützen. Daher fallen reine FrontEnd-Bibliotheken wie VueJS oder React aus der Betrachtung heraus.

3.5.1 *ASP.NET Core*

TODO: Anpassen auf "Kernpunkte"?

Vorteile:

- Gute Entwicklungsumgebung
- EntityFramework als Object Relational Mapping (ORM)
- Große Auswahl an Erweiterungen durch NuGet-Pakete
- Betriebssystem unabhängig

Nachteile:

-

Beim Vergleich zu JSP steht ASP.NET Core nicht hinten an. Im großen und ganzen ist der Funktionsumfang der gleiche und mit EntityFramework gibt es ebenfalls einen sehr mächtigen ORM. Hier sehe ich nur ebenfalls ein ähnliches Problem beim Caching wie bei OpenJPA, dass diese zu Problemen führen kann, wenn der Speicher voll wird.

C# wird ebenfalls wie Java in einen Zwischencode übersetzt und erst bei der Ausführung auf die Zielplattform übersetzt. Hierbei haben die meistens Test gezeigt, dass das .NET Framework hier um einiges Effizienter und

schneller arbeitet als die Java Runtime. Zusätzlich wird bei ASP.NET Core nicht noch ein zusätzlicher Server benötigt um die Anwendung aktiv zu halten.

3.5.2 *Golang*

Vorteile:

- schnelle und einfache Entwicklung
- Native Übersetzung der Anwendung (hohe Performance)
- Betriebssystem unabhängig
- Gut geeignet für Microservices

Nachteile:

- Neue Programmiersprache, noch wenige Programmierer

Schnelle und einfache Entwicklung durch das grundsätzliche Konzept der neuen Sprache Ist wiederrum aber der nachteil dass es noch nicht viele können. Ist aber wiederrum sehr schnell und effizient und für Microservices gedacht. Reiner ORM vorhanden, ohne weitere Caching und halten von Daten.

3.5.3 *PHP*

Vorteile:

- Text-Editor reicht

Nachteile:

- Script-Sprache

3.5.4 *Fazit*

TODO: Noch entscheiden, ob der Compare direkt bei der Technologie gemachtwird, oder allgemein am ende

PHP fällt aufgrund der Script-Sprache raus, da dies nicht verteilbar ist und an den Uni keine Entwickler zu finden sind. Bei Go sind es ebenfalls hauptsächlich die nicht vorhandenen Entwickler an der Uni ASP.NET Core ist im Vergleich zu JSP identisch, aber auch hier werden aktuell die Entwickler fehlen, da Java in den Vorlesungen dran genommen wird.

4

PERFORMANCE-UNTERSUCHUNG

5 OPTIMIERUNG

5.1 ERMITTLUNG DER PERFORMANCE-PROBLEME

5.2 ANALYSE DER ABFRAGE

5.3 OPTIMIERUNGEN DER ABFRAGEN

5.4 ANPASSUNG DER KONFIGURATION

und hier ein sql-beispiel Listing 5.1

Listing 5.1: ein sql beispiel

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

6 EVALUIERUNG

7

ZUSAMMENFASSUNG UND AUSBLICK

Teil I

APPENDIX

A

UMFRAGE ZUR OPTIMIERUNG

Herzlich Willkommen

Diese Umfrage ist Teil der Bachelorarbeit »Analyse und Optimierung der Webseite des Wedekind Projektes« von Marco Galster, die Rahmen des Projektes »Edition der Korrespondenz Frank Wedekinds als Online-Volltextdatenbank« an der Fernuni Hagen durchgeführt wird. In der Bachelorarbeit soll der aktuelle Prototyp auf Performance-Probleme untersucht und im Anschluss optimiert werden, um die Benutzerfreundlichkeit und die Akzeptanz der Anwendung zu verbessern. Dies soll dazu führen, dass die digitalen Briefeditionen verstärkt bei der Forschung zur literarhistorischen und kulturgeschichtlichen Wissenssteigerung eingesetzt werden.
TODO: der letzte Satz nochmal überarbeiten!

Der aktuelle Prototyp der Anwendung wird unter bereitgestellt. Die Fragen sind bitte im Rahmen ihrer normalen Tätigkeit an dem Projekt zu beantworten. Bitte geben Sie so viele Informationen mit an, die ihnen zu den Problemen mit auffallen.

Wir bedanken uns im Voraus für ihre Zeit und die Teilnahme an der Umfrage. Für die Umfrage benötigen Sie ca. 10 Minuten.

1. Welche Tätigkeit führen Sie aus? (Bearbeiter/Verwender)
2. Bitte geben Sie nun die Tätigkeiten an, bei denen immer wieder Verzögerung auftreten. Geben Sie bitte zu jeder Tätigkeit noch folgende Informationen mit an:
   - Wie häufig tritt die Verzögerung auf? (in Abhängigkeit zum Aufruf)
   - Gibt es einen zeitlichen Bezug? (z.B. tritt die Verzögerung nur Vormittags auf)
   - Tritt es immer einer bestimmten Abfolge auf, und wenn ja in welcher? (z.B. wenn man zuerst die Benutzerliste anwählt und dann in den Bearbeiten-Modus wechselt)

LITERATUR

[Ibm] 2023. URL: https://www.ibm.com/docs/de/was/8.5.5?topic=applications-configuring-openjpa-caching-improve-performance (besucht am 24. 09. 2023).

[Posa] 2023. URL: https://postgrespro.com/docs/postgresql/14/runtime-config-resource (besucht am 27. 12. 2023).

[Posb] 2023. URL: https://www.postgresql.org/docs/8.4/pgstatstatements.html (besucht am 27. 12. 2023).

[Posc] 2024. URL: https://www.postgresql.org/docs/current/runtime-config-resource.html (besucht am 27. 03. 2024).

[DNB21] Henrietta Dombrovskaya, Boris Novikov und Anna Bailliekova. *PostgreSQL Query Optimization - The Ultimate Guide to Building Efficient Queries*. Berkeley, CA: Apress, 2021. ISBN: 978-1-4842-6885-8. DOI: 10.1007/978-1-4842-6885-8. eprint: https://link.springer.com/book/10.1007/978-1-4842-6885-8. URL: https://link.springer.com/book/10.1007/978-1-4842-6885-8.

[EH13] Peter Eisentraut und Bernd Helmle. *PostgreSQL-Administration* -. Köln: O'Reilly Germany, 2013. ISBN: 978-3-868-99362-2.

[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.